SANDIA REPORT

SAND2005-0014
Unlimited Release
Printed January 2005

Graduated Embodiment for Sophisticated Agent Evolution and Optimization

Mark Boslough, Michael Peters, and Arthurine Pierson

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2005-0014
Unlimited Release
Printed January 2005

Graduated Embodiment for Sophisticated Agent Evolution and Optimization

Mark Boslough
Michael Peters
Evolutionary Computing & Agent Based Modeling Department

Arthurine Pierson
Intelligent Systems Principles Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0318

Abstract

We summarize the results of a project to develop evolutionary computing methods for the design of behaviors of embodied agents in the form of autonomous vehicles. We conceived and implemented a strategy called *graduated embodiment*. This method allows high-level behavior algorithms to be developed using genetic programming methods in a low-fidelity, disembodied modeling environment for migration to high-fidelity, complex embodied applications. This project applies our methods to the problem domain of robot navigation using *adaptive waypoints*, which allow navigation behaviors to be ported among autonomous mobile robots with different degrees of embodiment, using incremental adaptation and staged optimization. Our approach to biomimetic behavior engineering is a hybrid of human design and artificial evolution, with the application of evolutionary computing in stages to preserve building blocks and limit search space. The methods and tools developed for this project are directly applicable to other agent-based modeling needs, including climate-related conflict analysis, multiplayer training methods, and market-based hypothesis evaluation.

Acknowledgments

This project was funded by the Laboratory Directed Research and Development (LDRD) program under project 38610. Michael McDonald coined the term *graduated embodiment* and originated the concept, which itself evolved during this project. We thank Craig Jorgensen for help with parallelization, scripts, porting, and testing. Ben Hamlet, Kevin Oishi, and other members of the Umbra team assisted with code development and integration. This report was prepared with technical writing assistance from Rhonda Reinert.

Contents

Figures

Nomenclature

1D	one-dimensional
2D	two-dimensional
3D	three-dimensional
AI	artificial intelligence
DGGP	Distributed Generic Genetic Programming
DNA	deoxyribonucleic acid
DOF	degree of freedom
FDM	Flight Dynamics Model
GP	genetic programming
I/O	input/output
ISRC	Intelligent Systems and Robotics Center
LAN	local-area network
LDRD	Laboratory Directed Research and Development
LOB	level of behavior
MFD	modular-functional decomposition
MOUT	military operations in urbanized terrain
MPI	Message-Passing Interface
Sandia	Sandia National Laboratories
UAV	unoccupied (or unmanned) aerial vehicle
XML	Extensible Markup Language

Intentionally Left Blank

Graduated Embodiment for Sophisticated Agent Evolution and Optimization

1 Introduction

This research integrates two areas of focus at Sandia National Laboratories (Sandia): (1) development of genetic programming (GP) methods and (2) development of modeling and simulation environments.

The project has resulted in a new capability for generating reusable navigation behaviors by implementing adaptive waypoints and algorithms for graduated embodiment of autonomous mobile robots and swarms of robots. These algorithms can be evolved within the Sandia high-performance massively parallel computing environment. We developed a new object-oriented GP engine and integrated it with Sandia's Umbra modeling and simulation framework. These integrated tools can be used for a wide variety of simulation tasks (e.g., aerial reconnaissance, region mapping, and target search) that are needed to develop, analyze, test, and control systems and networks of intelligent machines. We intend to further develop these integrated tools and broaden the scope of their application to include other agent-based modeling and simulation problems. Such applications include agent-based analysis of conflict caused by environmental degradation, creation of behaviors and strategies in multiplayer games for training exercises, and development of trading agents and models for understanding and implementing information aggregation markets.

1.1 Initial Problem Domain: Autonomous Navigation

The tasks of trying to figure out where you are, where you are going, and how to get there are some of life's oldest dilemmas. Navigation, positioning, and path planning are crucial to virtually every activity undertaken by animals and humans, yet the process often seems impossible to describe, model, or prescribe. In the simplest examples, the goals and rules of motion allow for simple behavior algorithms that can be turned into human-generated computer programs that can control human-created systems. But such programs are brittle and prone to failure if the system does not behave exactly as anticipated.

The real world is full of contingencies, unexpected events, multiple (and often competing) goals, nonlinear responses, feedbacks, noise, and complex interactions across a wide range of time scales and levels of organization. Nevertheless, animals have evolved the ability to cope, prosper, and multiply in such a world. Their robust behaviors give them the ability to navigate and "plan" their path of motion in order to migrate, search for food, return home, find mates, and avoid predators. They are successful at these goals while simultaneously tending to lower-level tasks, adapting to changes in their environment, dealing with unfamiliar situations, and ignoring irrelevant information.

Human-designed autonomous mobile robots must possess similarly complex and adaptive behaviors if they are to be useful for the types of problems for which they are being proposed. Many of their goals are analogous to those of animals, and include migration, search, cooperation, exploration, location, and avoidance. In some cases, the adaptive behaviors of animals can be applied to the development of autonomous robots by a method we call "biomimetic behavior engineering" (Boslough 2002). Locomotion behaviors (such as dynamic soaring by albatrosses) can be analyzed and written into a control algorithm, but such purely "hand-coded" behaviors are usually unable to deal with the unexpected.

The simple borrowing of behaviors observed in the animal world and applying them to robotics is insufficient. The external manifestation of a behavior can be simulated, but the actual behavior includes the internal processing. Real animal behaviors tend to be bottom-up and emergent, not top-down and "hard-wired." One can write a computer program that will *appear* to code a behavior that, in reality, emerges from a complex nonlinear dynamic balance of different processes. Such programs can be useful for applications where the environment is predictable, but they amount to mimicry of external expression, not internal process. The "external" method of biomimetic behavior engineering is a form of "classical artificial intelligence" (AI) and suffers from the same limitations.

Our goal is to develop a paradigm for behavior engineering that generates robust, reusable, and useful control systems for autonomous mobile robots. Our primary application of interest is to develop goal-seeking navigation behaviors.

1.2 Application

Advances in mechanical and electrical engineering will continue to lower the cost of physical robotics. In battle-space applications, a single person controls one or more robotic entities. When the battle space expands and has hundreds or thousands of such entities, human control will be impractical for each individual robot and will shift to the direction of groups of robots. Individuals must act autonomously but work in a group to achieve a common goal. This research addresses relieving the human-in-the-loop not only during operation but also for design of the mission goal.

Reductions in the cost of robots and simulation platforms will permit the development of systems containing thousands of interacting agents. As groups or swarms attain large sizes, the number of potential interactions scales exponentially and provides both a greater opportunity to coordinate and a costlier penalty for interference. When the agents are mobile, navigation control algorithms utilize available actuators to move each of them through the environment effectively. Due to the diversity of physical environments and locomotion abilities, it is difficult to develop navigation control algorithms that generalize across entities and/or physical settings. Yet, navigation strategies such as "move-to-goal" and "avoid-collision" are already available. The limiting factor is balancing the resources, whether the entity is a virtual simulation robot or a physical robot.

For this project, we designed, developed, and tested a methodology and software tools for incremental adaptation, staged optimization, and graduated embodiment, and to address differing levels of behavior. We introduced a hybrid AI (artificial intelligence) approach that has many advantages for operating in a complex, dynamic environment. We believe that navigation strategies such as move-to-goal are reusable provided they are adjusted to suit the locomotion capabilities and degrees of freedom of each vehicle and its environment.

Using the generation of navigational waypoints as our example, we combined our efforts in GP (genetic programming) and modeling and simulation. Our test example is specific to navigation techniques. Navigation, positioning, and path planning are crucial to so many activities, and yet the process has always been quite cumbersome. Our development of the adaptive waypoint provides an interface between navigation control algorithms and the specific mobile entities on which they operate.

1.3 Overview

This report provides a brief summary of previous work on which it is based (Section 2), followed by a discussion of evolutionary design principles and the meaning of embodiment (Section 3). Section 4 provides a high-level presentation of the architectural principles that we believe address the problems of scalability, wide applicability, and competitiveness. We argue for behavior engineering based on evolvable building blocks and describe our methodological notions of incremental adaptation, staged optimization, and graduated embodiment.

This project was focused on the development, implementation, and testing of an object-oriented GP engine called Distributed Generic Genetic Programming (DGGP), which is summarized in Section 5. This code was specifically designed to allow us to work flexibly with the architecture and methodologies described in the earlier sections. We integrated the DGGP engine with Sandia's modeling and simulation framework (Umbra), which is described in Section 6. This allowed us to make use of the visualization and physical modeling capabilities that were already fully developed and provided leverage with many other projects.

For this report, we applied our methods to a robotic-glider benchmark problem (Section 7) that has been used for several earlier GP efforts. We discuss how our architectural and methodological concepts can be used to improve the performance of GP for that problem. In Section 8, we describe the novel concept of navigating with adaptive waypoints and explain how this method can be developed and extended to more general navigation problems using our concept of graduated embodiment.

In Section 9, we describe results that support the use of incremental adaptation, staged optimization, and graduated embodiment for enhancing the applicability, scalability, and competitiveness of GP for autonomous robots. We present sequences of screen shots that show the adaptive behavior generated by our integrated DGGP/Umbra tool, using adaptive waypoints to tackle the benchmark problem. In Section 10, we

provide a summary of the key aspects of our project. Section 11 identifies a number of other problem domains to which our tools and methods can be applied.

2 Previous Work

This project draws on many areas of research, so it is necessary to present the various topics outward from the fundamentals. To this end, we review the related literature by providing an overview of behavior generation for autonomous mobile robots with an emphasis on navigation within complex, dynamic environments.

2.1 Behavior Generation with Genetic Programming

Algorithms abound for behavior generation. Research in the planning and control of autonomous mobile robots has received much attention in the past two decades. We developed a simple yet effective methodology for composing goal-specific behaviors into a more general and capable reactive control system. Our method was motivated by a desire to avoid manual programming; thus we take advantage of automatic programming methods.

We chose GP (genetic programming) methods to develop our robotic behaviors because GP has been demonstrated to work for "proof of principle" problems. The book *Genetic Programming III* (Koza et al. 1999) documents 16 attributes that are needed for challenging a computer to solve a problem without explicitly programming it. No other methods come as close as GP, which currently unconditionally possesses 13 of the 16 attributes. GP starts with a high-level statement of what needs to be done; determines a basic sequence of how to do it; produces a computer program; automatically determines the size of the program; provides code reuse; provides parameterized reuse; determines internal storage needs; determines iterations, loops, and recursions; organizes into hierarchies; automatically determines architecture; implements a wide range of programming constraints; and operates in a well-defined way.

This research describes an approach where GP at least partially possesses the remaining three of the 16 attributes: wide applicability, scalability, and competitiveness with human-produced results. These remaining attributes are needed to produce the ultimate goal of the system for automatically creating computer programs to produce useful behaviors.

A variety of evolutionary approaches have been applied to the problem of developing navigational behaviors. Another application of evolutionary computing has been the discovery of algorithms for image processing. However, GP is usually quite demanding when computational load and memory usage are considered. This is particularly true when using GP to solve image-processing problems, where the fitness landscape can be extremely large. Our GP goal in this project is to take advantage of the power of GP in the set-up phase on a robot and still have it operate efficiently in real time.

We extend the work of Pryor (1998), Barnette et al. (2000), and Pryor and Barton (2004) who applied evolutionary methods to the development of robotic behavior. Pryor (1998) originally developed a GP model to solve a suite of high-level robotics problems.

The motion of Pryor's "robugs" was idealized and highly constrained so that the focus of his evolutionary model could be on high-level navigational behaviors and goals. Low-level maneuvering issues, such as locomotion, steering, and braking control, were not initially addressed. Instead, the simulated robugs were constrained to move on a discrete two-dimensional (2D) square grid with instructions such as "turn left," "turn right," or "move ahead."

Behaviors were automatically generated using a genetic program to solve a series of problems in which robots are randomly distributed on a grid with obstacles and rewarded for finding a source that is emitting a signal. The required high-level navigational behavior can be encapsulated as a computer program, which can be engineered by a variety of methods. In the simplest case—when effective rules are easy to conceive and implement—the behavior can be coded by hand. For trivial goals, common sense is all that is required for inventing rules. More-difficult challenges require more-sophisticated solutions, which can be generated by a variety of optimization methods, including thermodynamic analogy models, conventional guidance theory, or reinforcement learning.

Using a tree-based GP (genetic programming) representation for behavior (Pryor 1998; Barnette et al. 2000; Pryor and Barton 2002) allows a great deal of flexibility and can be adapted to many kinds of problems. The behavior programs execute by traversing a tree that is made up of elemental building-block units called nodes, which can either be a function or a terminal. Functions perform operations and contain pointers to other nodes. Terminals return values that result in an instruction to the robot. The trees themselves are generated by a GP model originally developed by Pryor (1998) and based on methods described by Koza (1992), within a general framework presented by Holland (1975). GP is a type of genetic algorithm, an evolutionary computing method that is based on the principles of biological evolution.

Artificial evolution takes place over many discrete steps called generations. Generations in nature are not synchronous because lifetimes and breeding times vary in length, but evolution has been in operation for billions of years on Earth. In the model, they are synchronized for simplicity (though when running in parallel, asynchronicity across processors is allowed), and the number of generations is limited, by practical considerations, to hundreds. Each generation consists of a population of individuals. In nature, these are organisms, and the population size can vary and can reach numbers of millions or billions. In the model, they are computer programs, and the populations are held fixed for a given problem, with typical sizes on the order of thousands. The Darwinian principle of "survival of the fittest" is applied. In nature, any individual that survives long enough to breed and generate offspring is "fit" by definition. In the model, each individual behavior program is assigned a numeric score based on a "fitness function" chosen by the researcher to represent the problem, and survival depends on rank. Finally, the individuals must reproduce. In nature, the genotype is carried by deoxyribonucleic acid (DNA), which is mixed between individuals by sexual reproduction, and random processes occasionally introduce mutations. The human genome contains between 30,000 and 100,000 genes that represent billions of base pairs. In the model, the "genotypes" of two individuals are mixed with a crossover operator,

and random mutations are introduced to allow exploration of other regions of the "fitness landscape" (see Koza 1992 for more-detailed descriptions of these concepts). These artificial genomes have much lower information content than DNA in nature, containing hundreds of genes represented by thousands of bits of information.

2.2 Autonomous Mobile Robots

An autonomous mobile robot in this work refers to a software construct situated in an environment in which it is capable of self-directing freedom in pursuit of its goals. (Weiss 1999). An embodied agent occupies a space in an environment and affects its surroundings. Embodied agents consist of (a) sensors used to observe their worlds, (b) a cognitive component used to plan an action, and (c) actuators (or effectors) used to carry out the action.

In a deliberative logic-based paradigm, mobile-robot control is divided into sense, plan, and act phases, with each phase handled respectively (Gat 1997). In our representation, "plan" equates to "compute" in a virtual world. The cycle is repeated at regular and very small time intervals throughout the agent's operational period (Figure 2-1). This paradigm works best for well-defined problems. Unfortunately, many of the environments where embodied agents might be used (certainly in most real-world environments) demand that the agents make decisions with alacrity due to the high rate of environmental change. As environments become more and more dynamic and complex, the problems become more severe, eventually causing the agents based on the simplest sense/plan/act cycle to become impractical (Brooks 1991).

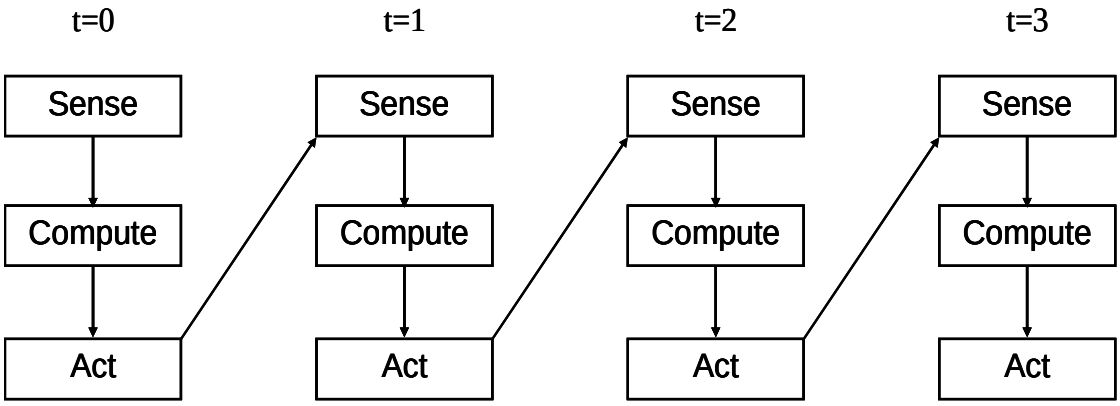


Figure 2-1. Sense, compute, and act.

Researchers have been developing new approaches in an effort to solve these very difficult problems facing autonomous systems. These related methods have been given many names by the researchers, including the animat approach (Wilson 1985), task-oriented subsumption architecture (Brooks 1986), computational neuroethology (Cliff 1991), and behavior-based AI (Maes 1993). Rather than explicitly planning, the reactive, behavior-based agents would simply reobserve the world at every computational step and act on what they perceive at that instant. The agent's decision-making function is

assembled from a number of simple behaviors that are triggered in response to sensory input. The simple priority-based behavior arbitration and lack of complex planning allow agents to react quickly to situations that demand it. However, while solving one problem by being adept at reacting to situations and real-time decisions, several other problems, such as scalability, inability to learn from past mistakes, and lack of cooperative behaviors, were introduced.

Despite the considerable history of progress in unoccupied aerial vehicles (UAVs), large repertoires of reactive skills have so far been elusive. This may seem surprising in view of the recent successes in implementing a host of specialist controllers capable of realistically synthesizing complex dynamics.

While a divide-and-conquer strategy is clearly prudent in coping with the enormous variety of controlled robotic motions that UAVs may perform, little effort has been directed at how the resulting control solutions may be integrated to yield composite controllers with significantly broader functionalities. The GP attributes needed (wide applicability, scalability, and competitiveness with human results) are also needed to produce useful robots.

We describe a hybrid approach that addresses the challenge of achieving an optimal (or at least acceptable) balance between reactive and deliberative behaviors. This hybrid approach differs from most existing hybrid architectures in two key ways. First, it uses a combination of fidelity levels. Second, it makes use of a vertical and horizontal hierarchical structure to manage the interaction between the layers.

2.3 Navigation via Waypoint Generation and Following

To test our combination of GP and hybrid AI methodology, we selected a robotic function of navigation. Path-planning strategies are very active research in the context of autonomous vehicles. Current literature (Meyer and Filliat 2003; Filliat and Meyer 2003) discusses numerous navigation strategies and outlines the adaptive capacities each uses to cope with complex, dynamic environments. Most algorithms must be tightly coupled with the vehicle that is used, including a real-time algorithm for combining waypoint planning and dynamic trajectory smoothing.

An example implementation (Beard and McLain 2003) in hardware also restricts the UAVs to flying at a constant altitude and to limiting the degrees of freedom to an autopilot that receives only velocity and heading commands. This research implements and quantitatively evaluates waypoint following as a means of high-level guidance to vehicles without regard to lower-level propulsion and physics or to middle-level autopilot algorithms.

Our method is also intended to allow us to borrow behaviors observed in the animal world by developing them under appropriate conditions. In other words, not to simply mimic behaviors by coding top-down hard-wired but to actually have behaviors developed bottom-up and emergent. We want to observe emergent behavior when individuals interact in the environment or in a group.

3 Biomimetic Behavior: Evolution, Design, and Embodiment

Embodiment includes the ability of a system to adapt to, learn from, and develop with its environment. Embodiment determines whether that system will "survive" in the environment. More-elaborate autonomous mobile robots require increasing computational effort that often proves too cumbersome and slow for real-world applications. The term, *graduated embodiment*, is the notion of a mobile robot adapting to appropriate fidelity ranges. A physically embodied agent must have

- the ability to coordinate actuator and sensor modalities to explore its environment,
- goal-oriented behavior on micro and macro levels,
- bidirectional interaction between the agent and its environment,
- bidirectional communication between the agent and other agents in the environment, and
- an understanding of the physics of the environment (Duffy and Joue 2001).

Biological evolution itself followed a path of graduated embodiment. Building blocks for biochemical pathways and structures developed during the billions of years before life was fully embodied, when it consisted of single-celled prokaryotes with limited sensory connections to a relatively stable, uniform, and predictable environment. Most of evolutionary history was spent developing the tools that would later be combined and recombined while graduating through higher levels of embodiment, from the first multicellular animals (metazoa) through *homo sapiens*.

Animat research attempts to follow closely with the thought that animals are created by biological evolution, not by a designer. Animals thrive under real-world conditions because their forms and behaviors are honed by billions of years of natural selection. Our modern understanding of the origin of animal behavior has its roots in Darwin's four basic postulates:

- Variation exists among individuals in a population.
- Some of these variations are inherited by the next generation.
- More offspring are produced in every generation than are able to survive.
- Individuals with the most favorable variations are the ones that survive and reproduce.

Consequently, survival and reproduction are not random. Favorable variations are selected by nature. Forms and behaviors that appear to have been designed are actually the result of Darwinian evolution.

Evolutionary approaches to behavioral engineering are biomimetic at a deeper level, because they mimic the adaptation that led to the structures and behaviors themselves. Evolutionary methods are inherently suitable for application to the graduated embodiment strategy in which high-level behavior algorithms that are initially developed using evolutionary computing methods in a relatively low-fidelity, disembodied modeling environment can be migrated to useful applications.

Evolutionary computing is appropriate for behavior engineering because it works with behavior building blocks that can be reused and recombined, much in the same way that biology operates. In biological systems, information is encoded in DNA, the raw material on which evolution operates. DNA holds the internal representation of an organism, its genotype. The outward manifestation of an animal, its phenotype, includes the reflexes, behavior, and intelligence that allow it to survive and reproduce as an embodied entity. "Internal" biomimetic behavior engineering emulates life in this respect and generates behaviors that emerge by operating directly on the genotype, which in this case is represented by computer code. When the code is developed by evolutionary methods, it depends on some degree of embodiment.

A drawback of artificial evolution is that it does not come close to the fidelity of evolution in the natural world in terms of numbers of generations, number of individuals in a population, or information content of the genome. However, the largest obstacle seems to be the fact that evolutionary computing methods are simulations that model "virtually embodied" entities, whereas natural evolution operates in the real world on actual embodied individuals. Unless the "off-line" simulation environment captures the important characteristics of the real world, successful behaviors may not survive when exposed to reality. In biology, on the other hand, the fitness of individuals has always been tested "on-line" under actual survival conditions.

It is not a profound observation that virtual embodiment is not the same as actual embodiment. The best argument for the physical grounding hypotheses of new AI is summed up by Brooks's (1990) remark that "… the world is its own best model. It is always exactly up to date. It always contains every detail there is to be known." But to evolve robotic behaviors from scratch in the real world would be impossible for lack of prokaryotic, self-replicating robots and billions of years. This is an extreme restatement of the fact that strong embodiment is not possible with current technology (Sharkey and Ziemke 2000). A realistic strategy for artificial evolution is to skip the impossible steps and graduate from virtual to physical embodiment through increasing levels of fidelity.

Natural evolution's embodiment graduated from simple to complex organisms. Only during the most recent ticks of the geologic clock did biological creatures possess the directed mobility, good eyeballs, and large brains that appear to be prerequisites for truly intelligent behavior (e.g., Moravec 1984). Technology already permits robots with these physical attributes, but they seem to be insufficient because of the lack of *history* in the software. Biological evolution is a historical process that takes place in a world where the future cannot be predicted. The DNA that encodes animal behaviors contains information built up over the entire span of geologic time. Successful behaviors are the ones that contributed to survival and reproduction throughout countless generations during times when the world may have been very different. The value of a particular behavior depends on how closely the present resembles the past. The ability of behavioral adaptations to emerge in response to changes in the environment depends upon rich and useful genetic variability, which results in part from history.

In the natural world, evolution simultaneously operates on an organism's morphology and behavior; there is no qualitative distinction between these two types of

adaptations. This is the approach taken by Sims (1994), whose virtual animats involve coevolution of both form and function. This strategy greatly expands the solution space that can be explored and allows the animat morphology itself to be part of the optimization process. Sims's animats are, however, still subject to limited degrees of freedom and constraints that are not imposed in the natural world. Moreover, the optimal solutions do not necessarily yield physical systems that could lead to an engineering design that would allow embodiment to be realized in the real world.

Engineering considerations require that hardware and software design be decoupled to achieve graduated embodiment. The alternative would be to redesign new hardware for every generation (with appropriate genetic variability) to reach an optimum. A more realistic approach is to develop building-block behaviors that can be applied across a broad range of hardware designs. As the design of the physical robot changes, a new behavior can be allowed to evolve from the pre-evolved building blocks. This method is a hybrid of engineered physical attributes and evolved behaviors, but takes advantage of "history" information embedded in the artificial genome that was accumulated during lower levels of embodiment and fidelity.

Some practitioners of evolutionary computing adopt a philosophy that "evolution should be allowed to operate on its own and eventually it will find the best solution." This hands-off approach attempts to bypass any human intervention whatsoever, presumably because human biases might prevent global best—but unanticipated—solutions from being discovered by machine. We reject this approach. It is impossible to carry out for two reasons. First is scalability; unlike the natural world, artificial evolution does not have access to billions of years, the high information density of DNA, or the large populations of individuals that can harbor a wide range of genetic variability. Second is the fact that artificial evolution by computer can never be free of human biases because evaluation functions used to rank the success of various phenotypes are written by humans and already contain biases.

Moreover, for robotics the hardware is also human designed. By its very definition, the problem is a hybrid that contains both designed and evolved components. There is already an interface between designed hardware and evolved software, and this interface must include designed software components, such as device drivers for sensors and actuators, to function. The boundary between designed and evolved components is arbitrary. For all these reasons, we have adopted what we view as the most pragmatic approach in which GP methods are used to extend, but not replace, the creativity and intelligence of the designer. There is no such thing as cheating by handwriting useful functions, but ultimately it is the "natural selection" that decides when to use designed building blocks and when to use those that have been evolved from scratch. The result is a hybrid of designed and evolved components at all levels of organization.

4 Architectural Principles

Our methodology can be applied to a wide range of applications far beyond goal-oriented navigation. The research develops a level-of-behavior (LOB) approach that addresses the challenge of achieving a balance between reactive and deliberative behaviors. It differs from most existing hybrid methods in three key ways because it makes use of *incremental adaptation*, *staged optimization*, and *graduated embodiment*.

We treat behavioral elements as building blocks that perform the compute function in the sense/compute/act cycle, as shown in Figure 4-1. We depict these behavioral elements as triangles to represent the tree structure of genetic programs, although hand-coded behaviors can be used as well. As building blocks, these behaviors can be combined both horizontally (for behaviors that operate either alternatively or simultaneously, resulting in one or more actions) and vertically (in which the output of one behavior provides the input to another).

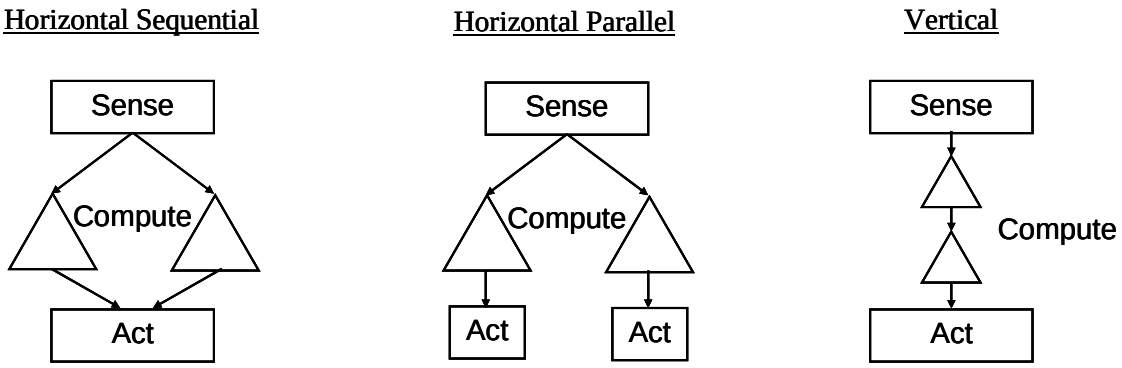


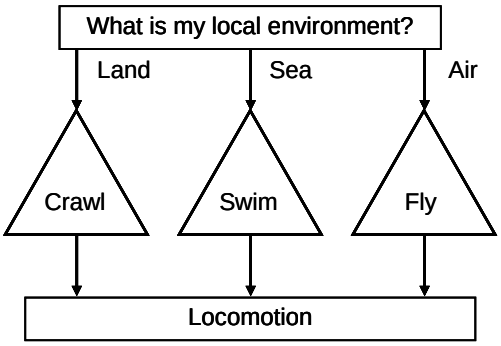
Figure 4-1. Classification of building blocks.

4.1 Evolution with Building Blocks

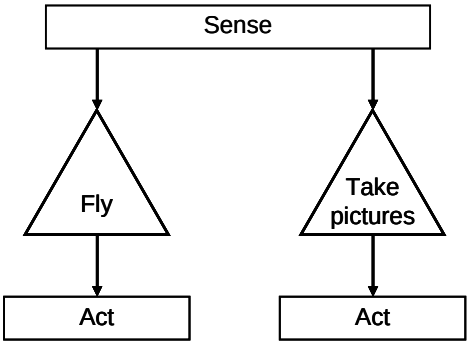
Our research makes use of building blocks in evolutionary modeling and simulation. For engineered software, the concept of building blocks allows various parts of code to be encapsulated according to function. Software that is not built in this way can easily degenerate into "spaghetti code" that is fault-intolerant and difficult to modify or improve. Likewise, if evolutionary methods are used to generate programs to accomplish large and complex sets of tasks, the resulting code is often inflexible and a poorly optimized dead end.

Building blocks can be combined both horizontally and in a vertical hierarchy, as in the examples of Figure 4-2. Behaviors can be assembled in a horizontal sequential integration when only one of the behaviors is called at a time. This could be a locomotion method that depends on the state observed by sensors, for example. Behaviors that serve independent actions (for example, locomotion and data collection) can be integrated in parallel. Finally, behaviors can be assembled vertically when one is a high-level behavior

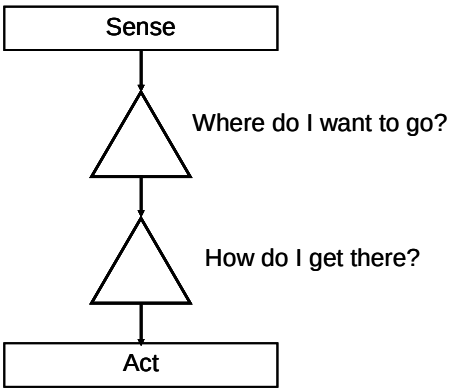
(e.g., navigation) and another is low level (e.g., locomotion). The most effective strategy is when evolved building blocks are augmented by human-engineered or handwritten building blocks.



(a) horizontal sequence integration



(b) horizontal parallel integration



(c) vertical integration

Figure 4-2. Examples of building-block integration.

Building blocks can be put together in more-complicated combinations, as in Figure 4-3.

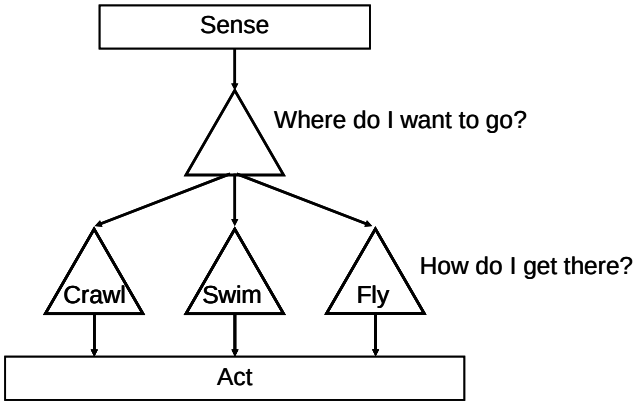


Figure 4-3. Example of combined building-block integration.

4.2 Incremental Adaptation

Incremental adaptation is a method by which building blocks can be taken from one situation and reused in a slightly different one. By successive readaptation to a sequence of environments or situations, a behavior's use can be migrated and reoptimized for other purposes. In effect, the fitness landscape can be varied on a time scale that is slow compared to that of evolution such that there is a selection pressure that forces the adaptation. This method is inspired by biological evolution in which adaptations that have one function can evolve to serve other purposes. However, for artificial evolution the change can be related to anything, including the level of fidelity or the number of degrees of freedom.

Modular-functional decomposition (MFD) is also a fundamental tenet of the field of computer science. We build MFD building blocks that are either computer-generated–based GP or hard-coded (e.g., 2D or 3D motion planner, autopilot). Each building block has deliberative behaviors when dealing with its environment via input and output connectors, as shown in Figure 4-4. These building blocks can be incrementally adapted to the level of fidelity (sometimes called the simulation level-of-detail). Rather than combining two building blocks, incremental adaptation provides a method to modify or enhance an existing successful behavior or to convert it to another use entirely. Individual components can be treated as "black boxes" encapsulating control knowledge. These components can be algorithms from technical fields such as biomechanics, robotics, or computer animation. The components are as simple or as complex as needed. For example, a hard-coded module could initially be quite minimal but enhanced with an incrementally adapted genetic program. Alternately, functionality can be restricted for real-time performance. In a purely GP module, successive generations can preselect more-sophisticated challenges as well as more-refined solutions.

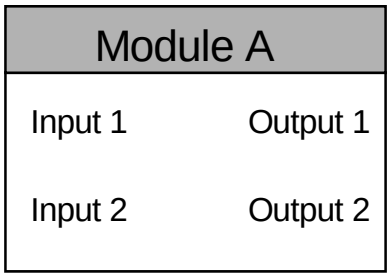


Figure 4-4. Single module with connector ports.

4.3 Staged Optimization

Staged optimization is a means of breaking a complex task into smaller subtasks that can be independently optimized. Nonscalability inherent in evolutionary methods is addressed by freezing some of the building blocks while others are optimized in succession, either independently or in coevolutionary sets. Our research concentrates on how the resulting solutions, either reactive or deliberate, may be integrated to yield composite controllers with significant broader functionality or useful applications. We build a behavior graph (as a vertical and horizontal hierarchical-control structure) to manage the interaction between the MFD and the interacting building blocks. Our graph provides for static parameters (defined as input/output [I/O] functions), dynamic parameters (defined as time-varying "instantaneous" values that map observations or sensations [input] to reactions [output]), and pointers to code containing algorithms for MFD building blocks and/or interacting building blocks.

In advanced implementations, security can be built into the system such that each behavior-graph value can have read or write privileges within the individual agent and for network access. Each LOB (level of behavior) of the building blocks can be switched on or off for varying levels of fidelity in the traditional style or can be cross-pollinated in the GP style. The behavior graph is not a simple directed acyclic graph like a scene graph in graphics. The behavior graph allows for feedback loops. Traversal of the graph is based on preconditions, postconditions, and expected performance. The graph also has provisions for the level of fidelity. Instead of combining all building blocks, staged optimization can allow the behavior engineer to concentrate on high-level behaviors before progressing to details. Staged optimization can also be combined with pruning to yield more compact code, as described in Section 7.1.

4.4 Graduated Embodiment

Graduated embodiment makes use of both incremental adaptation and staged optimization for the special case of evolution to higher levels of fidelity and embodiment. MFD is certainly a legitimate architectural principle, but it is hardly the only one possible. Reactive robotics has tended toward layered architecture with modifying functionality rather than toward the "black-box" organization. We build interacting building blocks that can either be computer-generated GP or hard-coded (e.g., collision

detection, trajectory smoothing). An interacting building block is one in which information is continually being sensed and behavior is continually being generated. Crucial to the distinction between an MFD building block and an interaction building block is the idea that interactive systems are always operating concurrently with their environment. Traditional computer procedures work from input at the beginning to output at the end. They are blissfully ignorant of their environment while executing. In contrast, an interactive subsystem is always responding to its environment. Each building block has reactive behaviors and is incrementally adaptive for the level of fidelity. However, by combining two distinct MFD building blocks, graduated embodiment provides the methodology to adjust the number of adaptable states and the scope of their perturbation with the environment in a time-varying constraint. We apply a specific definition of embodiment and degree of embodiment (Dautenhahn et al. 2001) to quantify our interactions. For example, the number of degrees of freedom might bear little relationship to an entity's ability to navigate, and we can use the same motion-planning or trajectory model, independent of vehicle specifications. Degrees of freedom can also be eliminated when vehicles move in unison with others or in swarms. Other degrees of freedom may need to be added to accommodate a robotic swarm.

For vertically integrated problems, some aspects are easy to build by hand, whereas others are very difficult or impossible when the model has many degrees of freedom. Our effective strategy is to mix evolved building blocks with hard-engineered or handwritten building blocks. Our intent is to evolve functionality that requires human and machine collaboration.

Rattlescape: A simple conceptual model can be used to illustrate these ideas. Suppose, for example, we wanted to design a behavior for a robotic rattlesnake whose only goal is to survive in an environment where it might be stomped by a large grazing animal. The human-designed snake hardware would consist of sensors and actuators. We want a snake that can sense, slither, and rattle like a real snake. A greatly simplified snake animat might have a dozen sensors and a hundred actuators. Evolution from scratch is impossible for reasons of scalability.

We know from human experience that a buffalo will avoid stepping on a buzzing rattlesnake, so we could write a driver by hand that activates the tail-vibration actuators in a cycled sequence at a frequency that creates the appropriate sound that will startle a buffalo.

Human observation also tells us that snakes exhibit several different modes of locomotion, most of which involve wavelike motion due to muscles contracting in a cyclic sequence. Armed with this knowledge, we may wish to reduce the number of degrees of freedom by considering, for example, symmetric and antisymmetric normal modes as the building blocks of locomotion, and allow artificial evolution to operate in a limited virtual world to select the phases and amplitudes that yield the most efficient "slither."

However, we might not know the best way to detect a large mammal, other than to suspect that the task requires some combination of vibration, heat, and visual-motion

sensing. The large-mammal detection algorithm might be evolved from much smaller sensor-data building blocks through a sequence of virtual worlds that model the coupling of ground motion to vibration sensors, and heat to infrared sensors.

We now have three algorithms with three different levels of organization and three different types of connections between their designed and evolved components. If these algorithms can be considered independent behaviors, another stage of artificial evolution can be used to find a way to combine them that leads to the best snake-survival rates. An obvious solution that would be quickly found by a genetic program would be that shown in Figure 4-5, which depicts a composite behavior tree that is constructed of three subtrees that are hybrids of sub-subtrees that have both designed (white) and evolved (gray) components at different scales of organization. In this case, near animals would activate the rattle behavior, and distant animals would activate "slither."

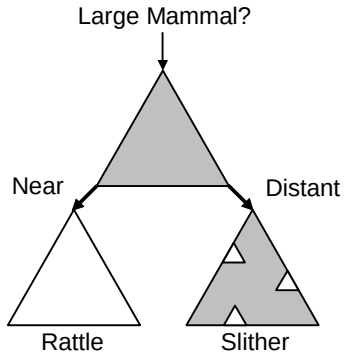


Figure 4-5. A composite behavior for "rattlescape."

Now suppose the world changes. The buffalo are gone and replaced by human beings who react to buzzing rattlesnakes by killing them. The same building blocks can be put together in a different way to generate a new survival behavior. In the successful snake animats, the detection algorithm now inhibits the vibration driver and perhaps activates the slither driver for all detections.

For this illustration we have made an assumption that will often fail in practice. We assumed that the three high-level behaviors operate in complete isolation. It may turn out that slither and rattle use common actuators and cannot be simultaneously activated. Another possibility is that the detection algorithm requires a stationary snake. Treatment of the behaviors as independent building blocks greatly reduces the number of degrees of freedom, but it may also prevent the discovery of new survival adaptations and higher-level optimization.

Our way of dealing with this is to apply staged optimization in which various high-level behaviors are evolved independently as in the above example. A global optimization is then performed by holding some behaviors fixed and allowing others to coevolve with the total system. This might lead to a new emergent behavior, such as "rattle and slither," if that is a positive survival adaptation.

5 The Genetic Programming Engine (DGGP)

Our new GP engine is named Distributed Generic Genetic Programming (DGGP) because it was intended from the beginning to be (1) parallelizable by distribution across networks and (2) of broad applicability for many GP applications. It was developed for this project as a generalized and object-oriented code that captures and extends the capabilities of an earlier Sandia GP code (Pryor and Barton 2002) with the added goals of modularity and integration with modeling and simulation tools. One of the DGGP engine's most important attributes is that it has been integrated with Umbra, Sandia's modeling and simulation framework (described in Section 6). This allows development to be compartmentalized, and code can be reused in a way that leverages both projects.

Development of the DGGP engine is continuing with funding by new projects, and full documentation with detailed specifications is in preparation by Peters et al. 2005. The DPPG engine currently runs on Linux (for clusters) and Windows.

5.1 Architecture

Sandia's DGGP engine is implemented using object-oriented programming techniques and contains a large number of classes. The core engine classes are separate from the model classes, so modeling and simulation programmers do not need to understand or even look at the GP engine when adding new models. The engine handles generation and evolution of the GP trees, file I/O, and parallelization. A large set of node types for the GP trees is currently available, and DGGP is designed so that users can easily add new node types.

The modeler is primarily responsible for developing a simulation that can accept the output of a behavior tree. For example, DGGP can be used to generate a behavior tree for navigation by returning the coordinates of a waypoint. Umbra has a C++ module for reading these coordinates. DGGP behavior trees can be used by Umbra without any additional programming by the many vehicles that can accept a waypoint as input.

5.1.1 Embodiment

The Embodiment base class currently represents an entity such as a vehicle or robot, but could just as well represent a nonplayer character in a training game. This class handles all interactions with the external world. The Umbra modeling and simulation framework contains a rich set of modules that we did not need to replicate within the DGGP engine. When working with Umbra, the primary purpose of the Embodiment is to provide an interface with the Umbra modules.

In some cases, the interface between the DGGP engine and Umbra must do more. For example, in our benchmark problem (Section 7), we defined a lift zone, which is an attribute of the external world that Umbra vehicles were never designed to recognize. To handle the lift zone, the Embodiment simply ignores Umbra's tracking of the altitude and

implements its own imaginary height. This flexibility allows us to run the benchmark problem on any Umbra vehicle without altering Umbra or the DGGP engine.

5.1.2 Individual, Behavior, and Node

The Individual, Behavior, and Node classes constitute the behavior trees (Trees). The DGGP engine allows a modeling and simulation programmer to get started using GP techniques without modification to these classes. Some of the responsibilities of these classes are in other helper classes, which are intended to be modified by DGGP users.

An Individual is a single entity with evolving behaviors. It is analogous to the brain of the vehicle or robot. Each vehicle has exactly one Individual, and each Individual contains one or more Behaviors. Each Behavior executes one stage of the decision process. For example, the first Behavior might select a distant waypoint as an ultimate goal, and then the second Behavior might select a close waypoint to seek as the first step in reaching that ultimate goal. The nearer waypoint coordinates can be sent to the Umbra vehicle, which uses an autopilot to do the actual driving. The component that interprets the Tree into a waypoint is in its own class, the ActionInterpreter, which is designed to be replaced.

A Behavior contains one or more Trees. It selects which Tree to execute based on the current situation. For example, it might have one Tree for ground, one for sea, and one for air. In another domain, it could have one Tree for offense and one for defense. The component that selects which Tree to use is in its own class (TreeSelector), so it can be altered without altering the Behavior.

A Tree is composed of gpNodes. Each gpNode performs one function, usually a fairly simple one, such as performing basic math, storing or retrieving data from a register, or checking for sensory data from the Embodiment. Each Node is implemented as a "`#define int`" in a switch statement, so adding a new Node type is easy.

5.1.3 ActionInterpreter

Input to modeling and simulation code varies depending on the problem and how it is implemented. The ActionInterpreter class is provided so that the decimal numbers returned by the Trees or contained in registers can be converted into a useful format that is used by the simulation. To accomplish the conversion, both the Behavior and the Individual have an ActionInterpreter, which is invoked after the classes execute. For our waypoint example, the ActionInterpreter takes the data in the first two registers and uses those as the coordinates *x* and *y*, respectively, of the waypoint. The component that interprets the Tree into an Umbra-style waypoint is in its own ActionInterpreter. If we want to change how the Tree is interpreted for another simulation requiring a different format, only the ActionInterpreter module, which is designed to be replaced, would need to be modified. The simulation can stay the same because the ActionInterpreter puts the data into a standard format. New Individuals will evolve to make use of the new format.

The ActionInterpreter class is optional. If this class is not specified, we simply use the decimal number returned by the Tree.

5.1.4 FitnessHandler

Fitness evaluation is particular to each type of run and can be changed during the analysis of a particular problem. The FitnessHandler class evaluates the fitness using data from the simulation run. It is also invoked at each simulation step to use data calculated by the simulation for use in the final fitness evaluation. For example, in the benchmark problem, the maximum distance from the origin was tracked. The final evaluation is to return this maximum distance (less some penalties, if specified).

5.1.5 Other Handlers

The RunManager, SimulationManager, and EvolutionEngine classes handle the run. The RunManager creates the population of Individuals and handles parallelization and output. Most of the functionality is in the parent class, so users can inherit from this to make their own runs by changing only a few functions, such as those used to set up the specific node types and helper classes that will be used in the run. Also provided is the QuicktestRunManager, which first performs a quick test and then performs a more extensive test on only those Individuals that have scored above a certain fitness. Users can inherit from this class and thereby get the quick-test functionality simply by specifying the two tests.

The SimulationManager handles the actual simulation. In general, it sets up the Umbra model, passes a set of Individuals to it, and runs the model for the specified time. To make a higher level of fidelity of a test (for example, changing one Umbra vehicle to another Umbra vehicle that models the physics more accurately), the SimulationManager is changed, and the RunManager is notified to use this new SimulationManager. The Umbra simulation run does not need to be modified when used in conjunction with DGGP.

The EvolutionEngine handles evolution of the Individuals. There are a number of genetic operators provided with the engine, such as crossover, mutation, and level crossover. The EvolutionEngine can execute multiple operations on a given Individual (for example, perform several crossovers, or first crossover then mutation). It also has multiple methods for selecting which Individuals to use in each operation. These classes have been designed so that users can easily extend the capability of the EvolutionEngine by creating new genetic operators and selection methods.

5.1.6 Parallelization

Coordination between processors in a parallel run is based on writing files to a common directory. The RunManager periodically outputs its best Individual and brings in the global best Individual. This parallelization step is asynchronous (each processor can import or export at any time, regardless of what stage other processors are executing), so blocking one processor while waiting for another to finish is not necessary. The

simulation step takes most of the time and is done locally on each processor. Evolution is also done locally. Consequently, most of the computations occur without any interprocessor communication. This method works across platforms and on both clusters and the local-area network (LAN). To date, parallelization is not implemented via Message-Passing Interface (MPI).

5.1.7 Integration with Umbra

The framework for connecting to Umbra has already been made. This works by giving the Umbra vehicle a waypoint to follow; that waypoint is the entity that the GP Individual produces. We have an Umbra module (ugp::WayQueue) included with the DGGP code that will take a waypoint from the GP Individual and give it to the Umbra vehicle. This module has been tested with multiple vehicle types. To use a new Umbra vehicle, a user need only make a new SimulationManager to set everything up.

5.1.8 Input to DGGP

DGGP has two types of configuration-file formats. The Extensible Markup Language (XML) format is recommended, while the earlier command-line format is still supported. The configuration files allow a user to change settings for the engine quickly without having to understand how the internal code works. Specifics on the options are described in the Examples directory, which is included with the source distribution, and in Peters et al. 2005.

5.1.9 Output from DGGP

The GP calculation can be restarted by reading in the globally best tree from a previous run. Trees can be written to a file in an ASCII format. The same file can be read by a simulator for validation and investigative studies or used to create an executable program on an actual vehicle. This ASCII file is based on the one presented by Pryor and Barton (2002), but with a generalized format.

Each file starts with header information about the settings for the Individual and the Behavior (type of ActionInterpreter, number of registers, etc). Following that, each node type is listed for use when creating new trees. Finally, the tree itself is represented by listing specific node information in the order of traversal. The ASCII trees are traversed by a method called depth-first ordering, and two human-readable formats, VERBOSE_BREADTHFIRST and VERBOSE_DEPTHFIRST, are available. Both formats change the numbers to actual words and add additional information, such as number of children, to the output. The difference is the ordering of tree traversal.

5.2 Creating a New GP Application

This section is intended to document the general concepts of the DGGP engine. Specific details are contained in the user's manual, which is being prepared (Peters et al. 2005). The following are the classes that a user must create for a basic new run using an already existing Umbra vehicle.

1 SimulationManager: This is the main class. It sets up the Umbra vehicle and therefore must be different for every vehicle.

2 RunManager: The RunManager must be set up to use the new SimulationManager.

3 FitnessHandler: If a user is doing a new type of test (not the benchmark), this class specifies how to evaluate fitness.

4 ConfigurationReader: The ConfigurationReader reads the configuration file and sets up the engine. The user must add a case to the switch that specifies the type of RunManager to create. This case will contain the code that calls the specified RunManager's constructor.

5.3 Running the GP Application

The major steps in executing a run are outlined below.

Step 1: Create initial population. This step is done based on the configuration file.

Step 2: Test Individuals. The RunManager passes the Individuals to the SimulationManager for testing:

A. At each time step, Individual selects an action:
   - Individual executes its first Behavior:
     - Behavior executes one of its trees, based on the situation (e.g., moving in air).
     - Behavior interprets the value of that tree into useful data, such as a set of coordinates to seek (waypoint).
   - Individual executes additional Behaviors. Each Behavior has access to the values from previous Behaviors (waypoint).
   - Individual interprets values from all Behaviors into an action (next waypoint to fly to).

B. Simulation advances one time step (handled in Umbra). Repeat A and B until simulation is complete

C. Assign Individuals fitness values based on performance in the simulation.

Step 3: Import and export Individuals to/from the global pool

Step 4: Perform genetic operations to make the next generation.

Step 5: Repeat from Step 2 until convergence.

6 The Umbra Simulation Framework

We developed a new modeling and simulation application, Adaptive Waypoints, because of the functionality it provides for general navigation problems. It was developed for this project as a specific application that extends the capabilities of Sandia's Umbra modeling and simulation code developed by the staff of the Intelligent Systems and Robotics Center (ISRC). The name, Umbra, means "shadow." The focus of the project was to build a simulation environment that augmented or supplemented the real environment. In other words, one learns by shadowing an expert. Sandia's Umbra modeling and simulation framework also makes use of object-oriented programming techniques.

This section provides a high-level description of Umbra's major components and features. Umbra development is continuing with funding in fiscal-year 2005 from numerous lab-wide projects, including participation in three Grand Challenge LDRD projects. Umbra is currently running on Linux (for clusters) and Windows.

6.1 Modularity

Umbra is a framework, a basic system model for a particular application domain within which specialized applications can be developed. Because it inhabits a particular domain, Umbra does not suffer from excessive abstraction. Umbra is a simple modular, composable set of building blocks that is conducive both to writing simulation components from scratch and to leveraging existing external simulation capabilities.

Modules are complete packets of code that perform the mathematics behind a simulation and are a core component of C++ programming. Modules provide flexibility and allow designers to divide up simulation tasks and attributes into manageable "chunks" and then connect them to share data. Any connections between modules should be "safe," or of the same type according to C++ rules. Connections can allow forward or feedback data exchange and can include parameters.

6.2 Additional Features

Some additional features to make developing a new application faster and easier have been implemented and are summarized below. Full documentation of Umbra is available at http://umbra.isrc.sandia.gov.

6.2.1 Scripting Capability

Umbra includes a dynamic link library, which contains a compiled C++ class, and "interpreted" wrapper libraries specifically for Tcl/Tk. Extensive error and information messages are available to help the user compose commands.

Umbra is totally flexible due to its component-based approach. Looking at several frameworks from different application domains from the flexibility perspective, we

notice that, regardless of the application domain, these frameworks form a continuum between two extremes, one represented by fully compiled systems and the other by fully interpreted systems. Figure 6-1 shows the relationships among framework flexibility, implementation complexity, run-time performance, and the compiled/interpreted function incorporated into the framework. With Umbra, the user can choose the right balance between compiled and interpreted code for a specific application and programming skill.

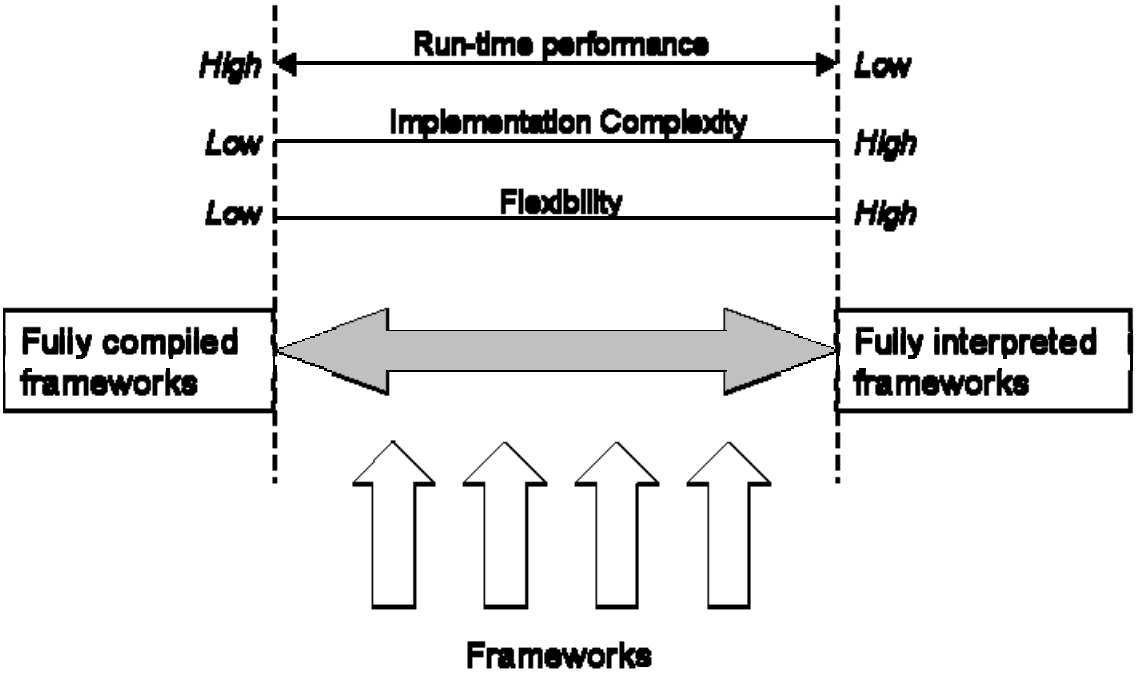


Figure 6-1. Compiled versus interpreted frameworks.

Scripting allows both dynamic instantiation (the insertion of software modules or even hardware-like interfaces to robots) and computational steering of the simulation during run-time. Examples include creating and deleting modules, modifying module parameters, and connecting and disconnecting modules.

6.2.2 Focused Initiative

To keep the task to a manageable scope, the software designers decided not to design Umbra as an all-encompassing simulation program. They chose instead to create a modular, script-capable framework that would be conducive to creating simulation components from scratch and to interfacing with external simulation capabilities. In building this focused initiative, it was extremely important to interface properly with other graphics and simulation efforts. The solution was a way for modules to be grouped with other modular components called metamodules, as shown in Figure 6-2.

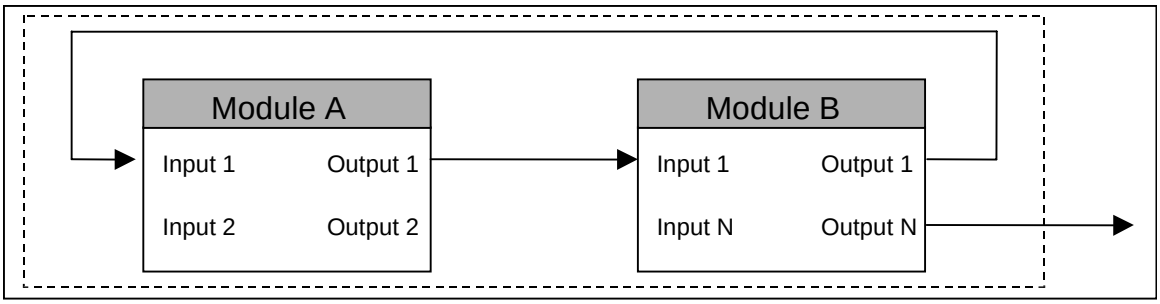


Figure 6-2. Umbra metamodules.

Modules have input and output connectors, much like real hardware, that allow data exchange between them (including feed forward, feedback, and parameters) and that form a data-flow network. The user programs the connectors, and Umbra tracks their use and state. As mentioned previously, any connections between modules should be "safe," or of the same type according to C++ rules (e.g., int, real, Boolean, etc.). It is important to remember that connectors are defined during project development but are not actually connected (using the hardware analogy) for data flow until run-time.

6.3 Components

An Umbra user can be any researcher of any discipline who has 2D or 3D data on which they want to perform simulation-based design and analysis.

Umbra is actually made up of multiple building blocks, each of which contributes to a modeling and simulation framework. Umbra can be a tool for users who want to use its turnkey systems, as well as for developers who desire more control over the ModSim process (Figure 6-3).

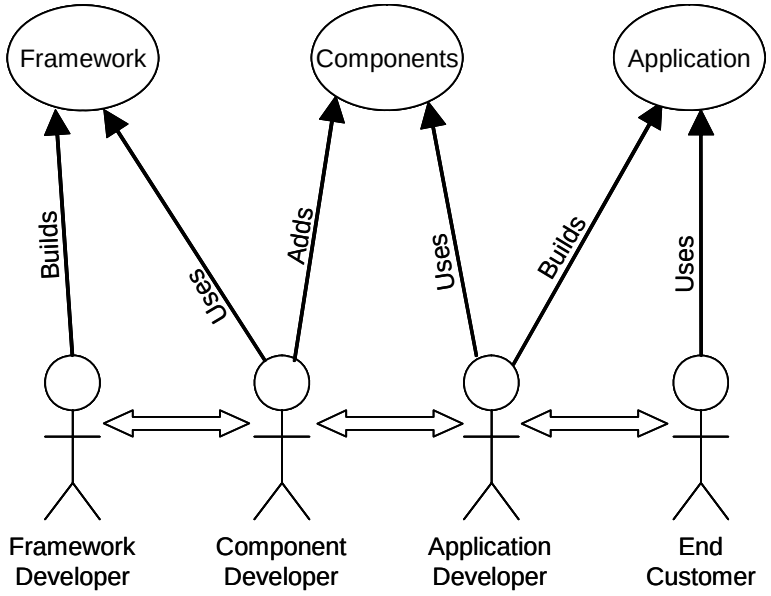


Figure 6-3. Umbra user community.

As defined below, four roles are important considerations in a framework design and are provided in the Umbra framework. For example, there will be a requirement for a component-building facility if the framework is to be a flexible and useful tool.

1. Interactive Umbra – An end customer is immediately able to use Umbra with this complete turnkey application, the Umbra Interactive Development Engine. It allows the user to immediately build simulations by using predefined modules.

2. UmbraGE – An application developer begins customization through the graphical interface system, the Umbra Graph Engine. It takes away many of the C++ nuances of inheritance and templating necessary to generate complex behaviors in a simulation. This allows a user to concentrate on behaviors in a specific field of expertise without having to become a computer science expert.

3. Umbra Kernel – A component developer's toolkit, the Umbra Behavior and Simulation Engine, is the base domain-specific functionality of Umbra. It provides a set of related and reusable C++ classes designed to provide general-purpose functionality to add behaviors to simulations. The Umbra kernel consists of classes compiled into a dynamic link library. Functionality included would be the basic data structures to define a scene graph, a behavior graph, a data-flow paradigm, and initial physics algorithms.

4. Umbra Framework – The complete environment for building modeling and simulation environments is often referred to as Umbra. The Umbra framework has been used for analysis of real-world problems. Umbra is continually adding more functionality to complete the computer-modeling schematic hierarchy from contributions such as this project. These components are add-on features that may or may not be relevant to a particular application domain. However, if work such as this project is useful to other applications, the modular architecture allows for easy reuse of modules.

6.4 Input to the Program

The basic Umbra program accepts numerous data files. Typical data modules such as terrain, autonomous mobile-robot representations, obstacles, and environmental conditions would be loaded in conjunction with the evolved LOB (level-of-behavior) graph to implement an application. The appropriate module to be loaded is based on the three-character suffix of the data file. For example, the LOB graphs from the genetic program end in ".xml".

6.5 Output from the Program

Umbra is designed for analysis. With the focused initiative, Umbra's strength is physics-based simulation, but it has been extended using its modular structure with this GP effort for behavior-based situational awareness. In general, interface modules are available for user interfaces, a graphical interface using Open Scene Graph, collision

detection, and numerous other modules such as communications and terrain following. All modules are optional. For example, output using the graphical interface and OpenGL is not mandatory but available.

6.6 Parallelism

Modularity has many benefits, but it cannot address the issue of large-scale phenomena (e.g., communications, different types of physics math, interfacing with different simulation programs). Modules need to operate in the world scope either completely within the Umbra framework with other modules or by interfacing with other frameworks with completely different administrations. World modules can spawn interactive “child” modules and have a functional relationship with them. World modules could also provide heterogeneity, the ability to integrate with different simulation programs or programmable objects on the same machine or networked machines running in parallel.

7 Benchmark Problem

As a benchmark for implementing and testing our various concepts, we continue to use the grid-based UAV (or robotic bird) problem defined by Pryor and Barton (2002). The problem is related to the robug problem in that the individual (sometimes called "robird" and henceforth called "glider") is constrained to move on a 2D Cartesian grid. It requires flight-behavior logic to search for uplift regions and investigate the surrounding area without crashing. This problem is more complicated than the robug problem because of the different procedures that must be performed sequentially. The agent must search and map the uplift region before the surrounding area can be explored, and conflicting goals of exploration versus exploitation (of the lift zone) must be balanced. This conflict was too difficult for a conventional genetic program, and more advanced methods had to be added. Additional features were also added to the benchmark problem to separate behavior functionality from a specific entity type (i.e., robug, glider, plane).

A rectangular "lift zone" is generated with its short dimension fixed at 20 units and its long dimension randomly chosen from a value range of 85 to 105 units. The long axis is randomly oriented along the x, $-x$, y, or $-y$ axis of the Cartesian coordinate system. The rectangle is placed such that the origin is five units from one end and centered on its width. This leaves between 80 and 100 units extending in one of the four directions. A glider is placed with its orientation chosen randomly from one of the four directions at a random location within a 100-unit-square box centered on the origin, at an altitude of $|x| + |y| + 20$ units. When it is outside the lift zone, the glider loses one unit of elevation for every time step. Inside the lift zone, the glider gains one unit, up to a maximum of 250 units. It moves one positive or negative unit along either the x- or y-axis every time step. The glider can go straight, turn left, or turn right, or make a U-turn in place. The fitness is defined by the maximum distance away from the origin that a glider reaches before returning to the lift zone. The actual fitness is associated with the evolved behavior program, not with a given instantiation of the glider, so the fitnesses of individual gliders running the same program—but with a range of initializations—are averaged. If its altitude goes to zero, the glider crashes and dies, and its contribution to the fitness function is assigned a large negative value.

This would appear to be an extremely simple problem compared to actual flight, but it involves staged and somewhat contradictory goals that force a balance between exploration and exploitation. The conservative glider will stay in the lift zone to preserve its life, but will get a low score. The bold glider will fly away from the lift zone and increase its risk of crashing. The highly ranked glider will require a delicately tuned algorithm. The benchmark problem also involves a set of goals that must be accomplished in sequence: (1) the glider must find the lift zone, (2) the glider must exploit the lift zone, (3) the glider must explore away from the lift zone, and (4) the glider must return to the lift zone.

To tackle problems with this level of complexity, we developed our adaptive-waypoint-following method as a means of high-level guidance to vehicles without regard

to lower-level propulsion and physics or to middle-level autopilot algorithms, as described in this paper in Section 8.

7.1 Staged Optimization and Pruning

For this type of problem, the behavior-optimization process is much more efficient when a complex task is broken into smaller subtasks. In the benchmark problem, the initial genetic flight-control algorithms that have been developed involve mere thousands of operations and can be described by a "genome" of a few thousand bytes. A simple doubling of the number of tasks that the behavior must manage—or a similar increase in fidelity—would put the problem out of reach on most compute clusters. Building-block efficiency seems to work best when the various subtasks are independently optimized in parallel by using a "policy table" that can be expressed as a root node with a set of pointers to the various behavior trees.

Boslough (2002) showed that staged optimization led to the fastest improvement in evolved behaviors but required more intervention. In the staged process, one or more subbehaviors can be locked, while the remaining trees are independently optimized. Figure 7-1 depicts this process, in which the white tree is a designed policy table, black trees are locked, and gray trees are undergoing evolution. This procedure greatly reduces the parameter space to be searched by the program.

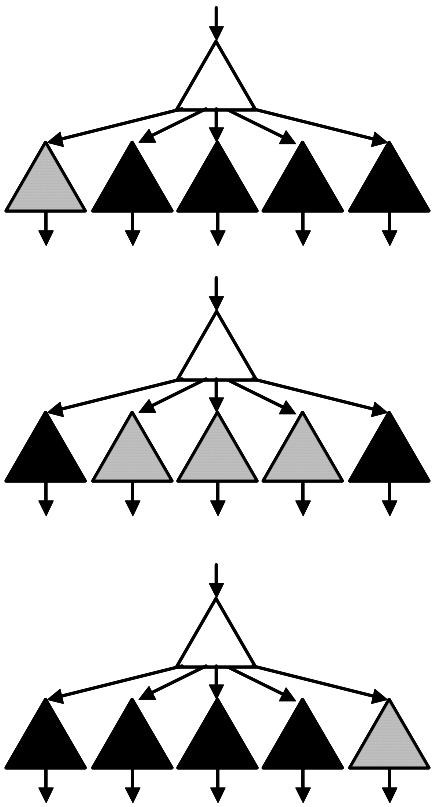


Figure 7-1. Staged optimization reduces search space.

Combining staged optimization with sequential pruning of the behavior trees leads to the fastest improvement. In sequential pruning, the behavior code is reduced in size by allowing the genetic program to remove superfluous parts by applying a size penalty to the fitness function. Figure 7-2 depicts this process schematically by representing small (pruned) behavior trees with small triangles. By locking all trees except the one to be pruned, we can accelerate this process.

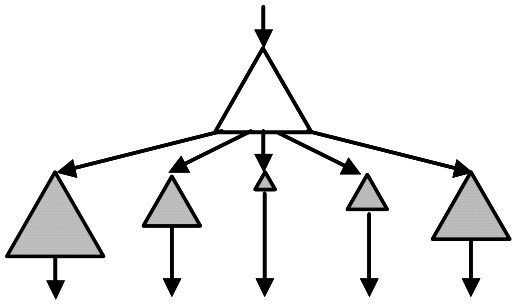


Figure 7-2. Pruning of trees removes useless code.

We recognize that some complex real-world behaviors will be much more difficult to break apart into smaller units, and in other problems the subtasks are not independent and must be optimized together. We think that by using building blocks we can allow the behavior designer to make the decision as to what constitutes a subtask, and to provide flexibility as to which sets of subtasks are most effectively coevolved.

8 Navigating with Adaptive Waypoints

Integrating behavior algorithms in a hierarchy is also required to create a comprehensive behavior system that includes cognitive, instinctual, and reflexive levels of behavior in addition to physical model attributes. In one example of vertical integration, we implemented adaptive waypoints to provide a means of high-level guidance to vehicles without regard to lower-level propulsion and physics or to middle-level autopilot algorithms. This method is shown schematically with pseudocode labels in Figure 8-1. Adaptive waypoints can be used to capture goal-oriented and collective agent behaviors that can be transferred from one type of vehicle to another as a self-contained behavioral "building block." Each individual vehicle is guided by its own dynamic waypoint, which moves in such a way as to lead the vehicle toward its individual or collective goal. The trajectory of each vehicle is determined by its flight dynamics model (FDM) and its autopilot. The FDM can be represented by simple nonphysical rules of motion, or it can be a full 6-DOF (degree-of-freedom) aerodynamics model.

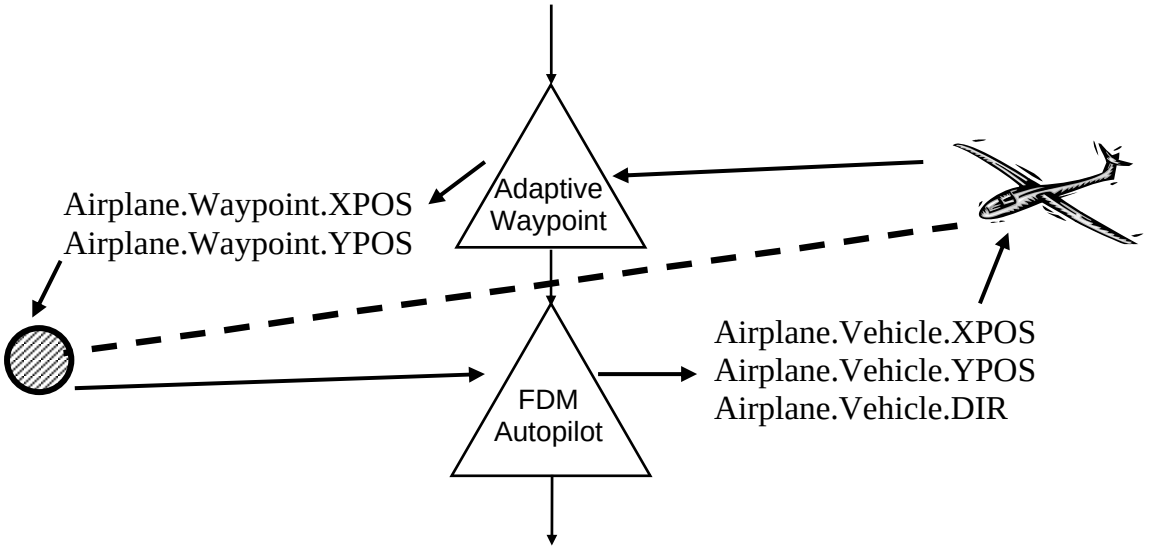


Figure 8-1. Schematic relationship between adaptive waypoint and FDM/autopilot.

Waypoints are coordinates in a virtual world. An entity can follow waypoints in the order they are given (e.g., read coordinates from a file) or an entity can be assigned to perform certain tasks (e.g., loiter) once it reaches a certain location. Waypoints can be either user-generated or entity-generated, and they can be static, dynamic, or adaptive. A waypoint can be thought of as an internal representation or as part of an agent's cognitive map.

Waypoints can take on various attributes of longevity, type identification, and goal orientation, as explained below.

Longevity. Static waypoints are intended to be permanent navigational aids and can be stored. Dynamic waypoints are intended to mark locations that are of temporary tactical importance (e.g., the last known location of an enemy). While

static waypoints remain in the entity's memory queue indefinitely, dynamic waypoints have a clearly defined lifespan. After the lifespan is up or the fade factor expires, the dynamic waypoint expires and is no longer stored by the entity. Adaptive waypoints can either be static or dynamic, but their longevity is determined reactively since they respond in a timely fashion to changes in the environment. Adaptive waypoints are determined by continuously running processes.

Type identification. All types of waypoints can be tagged with types to identify what kind of behavioral information they represent or any other information in addition to position. Static and dynamic waypoints have fixed type identification. Adaptive waypoints are autonomous since they exercise control over their own actions and can take on new identities when needed. Identity can be learned and changed based on previous experiences.

Goal orientation. Often, waypoints are used to solve situational goals. Static waypoints have no goal orientation. Dynamic waypoints contain information that may aid others in obtaining a goal. Adaptive waypoints do not simply act in response to the environment but are goal oriented and can communicate with other agents to determine a goal.

The intelligence of the waypoint can be transferred from one type of vehicle to another as a self-contained behavioral "building block." Each individual vehicle is guided by its own adaptive waypoint, which moves in such a way as to lead the vehicle toward its individual or collective goal. The intelligence of a vehicle can be contained entirely within the adaptive waypoint that is aware of and can respond to the state of the vehicle, the vehicle's environment, its high-level goal, and (in the case of collective behavior) the state of neighboring vehicles.

At every time step, the vehicle's position is updated based on the position of its waypoint, on its state, on its dynamics model, and on the autopilot system whose goal is to take it to the waypoint. The waypoint coordinates are updated based on the vehicle's behavior algorithm. The output of the algorithm depends on the location and direction of the vehicle, but not on the details of the internal state of the vehicle.

Behavior algorithms for adaptive waypoints can be developed using the methods of GP and graduated embodiment by staged optimization, as highlighted in Figure 8-2. The advantage of graduated embodiment is that behaviors can be evolved in the absence of a high-fidelity vehicle model, reducing the computational cost. Basic waypoint behaviors can be developed using low-fidelity vehicle models that only approximately reflect the actual vehicle performance. Once a waypoint behavior is developed for the low-fidelity vehicle model, the behavior can be tuned through successive stages of higher fidelity until the highest fidelity model is achieved. This is the essence of graduated embodiment.

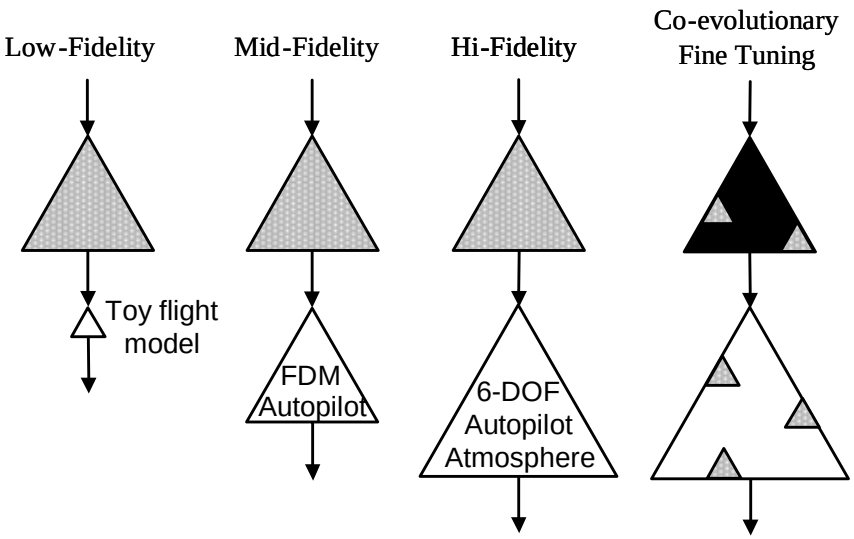


Figure 8-2. Graduated embodiment of adaptive waypoint.

Example problem: We can illustrate these concepts with a stripped-down problem that shows that a simple adaptive waypoint can be graduated from a 1D agent to a vehicle that moves in two dimensions. Suppose we have a vehicle (V), a target (T), a fixed waypoint (FW) an adaptive waypoint (AW), and a home (H). Let the initial position of V=0 and FW=100. T is a random number between 1 and 200, and H=0. The position of AW at every time step is entirely determined by the genetic program.

For the simple world, the rules of motion allow V to take one step in the direction of AW every time step. The rules of detection are that V detects T when they have the same coordinate. To be successful, V has to locate any target between 1 and 100 and return to H. The evaluation function is such that the most-fit behavior is the one that completes the goal in the shortest time, averaged over some number of tests.

The behavior can easily be coded by hand, but our genetic program discovered it within the first few iterations: The AW is set at infinity until the vehicle reaches either T or FW, and then AW is set to negative infinity. When the vehicle is allowed to explore in a 2D plane, with a constraint on its turning radius but otherwise following the same rules of motion, the adaptive waypoint no longer works because the vehicle does not return home (because of the offset due to its turning radius). A single mutation from the first solution will, however, solve the problem. The final AW location evolves from negative infinity to zero.

The point of this illustration is that in going from the simple to the more complex, many parts of the problem may already be solved, and those behaviors can be incrementally evolved in a way that allows building blocks and information to be reused.

9 Application of Adaptive Waypoints

The results show our progress in developing the methodology and tools for adding the three components that GP methods have been missing from the 16 attributes of Koza et al. (1999) listed in Section 2.1: wide applicability, scalability, and competitiveness with human-produced results. These remaining attributes are needed to produce the ultimate goal of the system for automatically creating computer programs to produce useful behaviors. This section gives a few examples of progress and summarizes the application of adaptive waypoints to the benchmark problem.

9.1 Wide Applicability, Scalability, and Competitiveness

According to Koza et al. (1999), systems for automatically creating computer programs should yield entities that have wide applicability. Automatic programming methods should produce solutions to many types of problems in a variety of fields. In one effort to address the applicability problem, we developed two different interaction building blocks: search and navigation. Figure 9-1 shows the glider problem fully integrated into an Umbra modeling and simulation application (in subsequent figures the complex terrain reader—another example of an Umbra building block—has been removed and replaced with a simple checkerboard reader to highlight the gliders more appropriately).

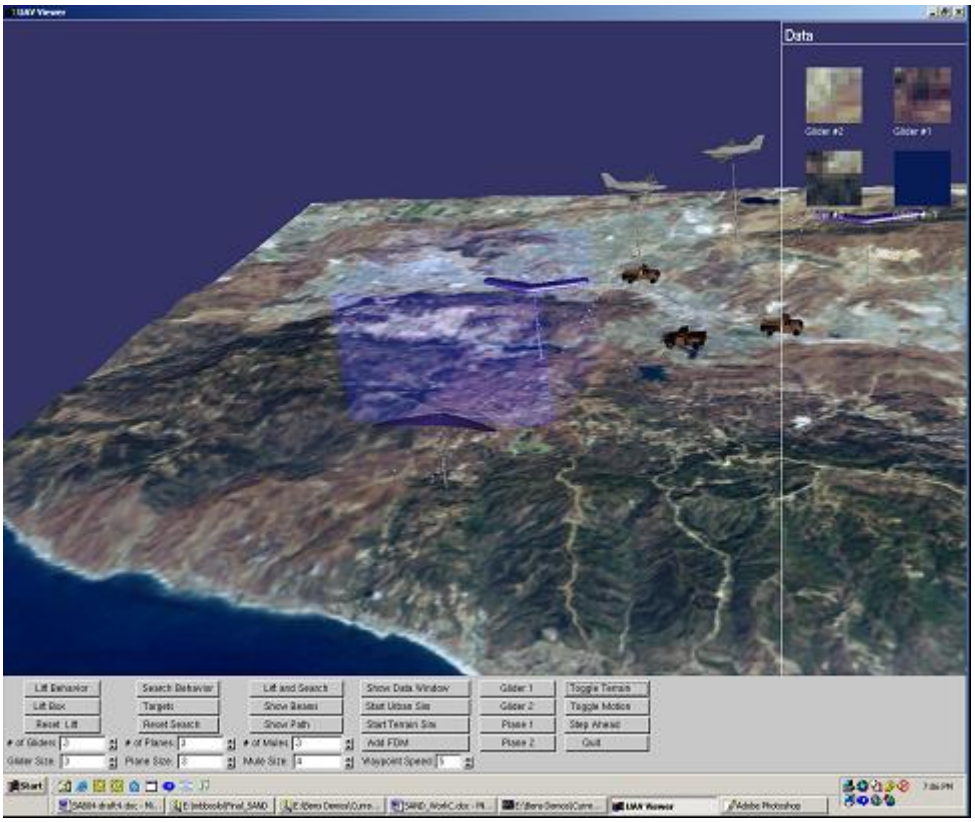


Figure 9-1. Sample application combining navigation and search behaviors.

In this case, we concentrated on establishing a framework for composing behaviors and integrating these behaviors into a graphical interface for a demonstration. We were able to show that independently evolved navigation and search behaviors could be combined as building blocks. Much work remains to be done in this area, but we believe that this integrated modeling and simulation framework will provide useful tools for combining and observing evolved behaviors in a wide variety of applications.

Scalability of an evolutionary process means that it scales well to larger versions of the same problem, according to Koza et al. (1999). In our problem domain, higher fidelity versions of the same problem provide the primary scalability hurdle. We believe that scalability depends on the individual's ability to exploit functional modularity, structural regularity, and the recursive composition of function and structure into increasingly larger, complex domains. We addressed scalability by implementing incremental adaptation and staged optimization techniques. We have not yet demonstrated scalability to high-fidelity problems, but we have developed the tools to do so in a way that can be quantified.

Competitiveness with human-produced results is probably the most difficult attribute to develop, quantify, and demonstrate. This requires that the method produces results that approach those designed by human programmers and engineers. Again, we have developed tools that can be used to directly compare human-coded behaviors to evolved behaviors.

9.2 Application of Adaptive Waypoints

To test and illustrate the capability of adaptive waypoints, we defined a simplified version of the benchmark problem. This reduced-complexity problem allowed behaviors to evolve sufficiently fast that they could be run and observed in real time on a desktop computer. In this simple version, the lift zone is always aligned along the same axis, and all gliders are initialized with the same position, orientation, and altitude. This test problem is much easier for the genetic program to solve than is the full benchmark case.

Figure 9-2 shows one time step from a simulation of a first generation of gliders, before they developed any interesting behaviors. In this case, all behavior trees were randomly generated. The checkerboard surface is the zero-altitude plane, and the squares have a dimension of 200 units. The entire problem domain is 2,000 units square, and the waypoints are constrained to remain on that surface. Waypoints are represented by small green balls, and their associated gliders are connected by the white tethers. The lift zone is the shaded rectangular volume. The origin is at the right end of the lift zone in this view. Many of the waypoints (and associated gliders) are clustered there because the registers that determine the location of the waypoint are arbitrarily initialized to zero (this is not a requirement and can be changed if desired). Most of the other first-generation gliders are following their waypoints to distant locations, so they will crash and get a fitness penalty. Note that, unlike the original benchmark problem, these gliders are not constrained to the four compass directions; they have more degrees of freedom.

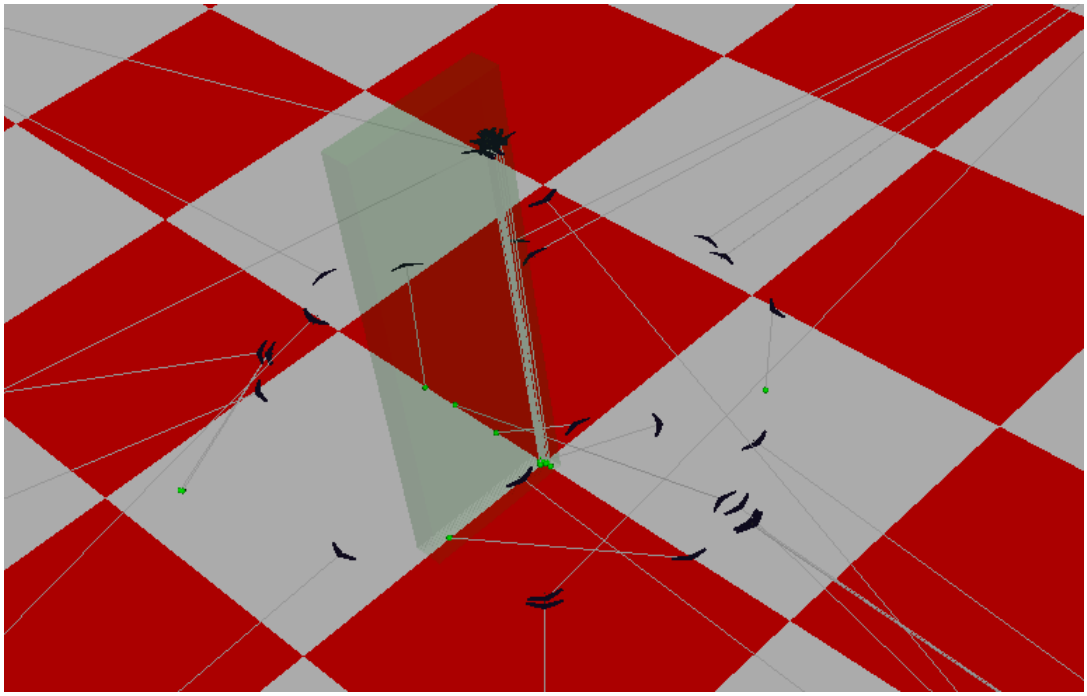


Figure 9-2. Untrained, first-generation gliders.

In Figure 9-3, a sequence of time steps from a later generation is shown. Within about a half hour of single-processor computing time, interesting behaviors have begun to develop.

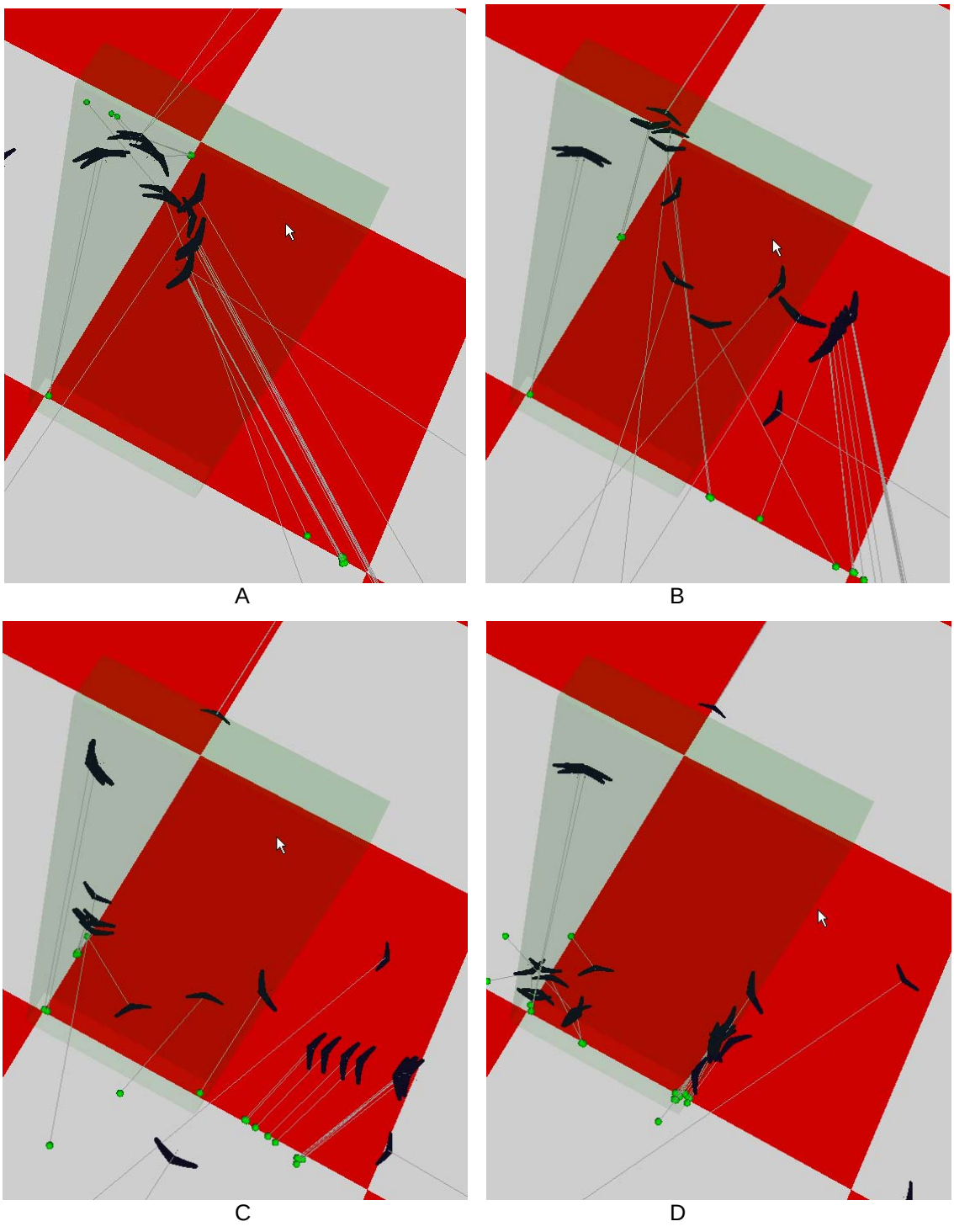


Figure 9-3. Sequence for later generation showing adaptive behavior.

The frames in Figure 9-3 are reoriented to place the origin at the left end of the lift zone. The most-fit gliders have learned to fly along the long axis of the lift zone. In the first frame (A), they are pursuing their waypoints, which have been placed beyond the end of the zone by their GP trees. In the second frame (B), most of them have left the lift

zone and are descending as they approach their maximum distance from the origin. In the third frame (C), they continue to descend, and their waypoints are returning to the lift zone as their waypoints attract them back in that direction. In the fourth frame (D), the most-fit gliders are now circling and loitering just within the lift zone so they can stay alive and get credit for the long-distance run they made in the earlier time steps. Meanwhile, the lower-fitness gliders have crashed and gotten a negative fitness score, or they are merely circling the origin to achieve a small positive fitness. It is the set of high-fitness gliders that will be preferentially selected for mutation and crossover for the next generation.

9.3 Incremental Adaptation of Waypoints

We can use the simplified version of the benchmark problem described in Section 9.2 to illustrate incremental adaptation using waypoints. We outlined a process for graduated embodiment such that a behavior can be evolved cheaply by using simplified rules of motion that have low computational cost. As more sophisticated and expensive rules of motion are applied (e.g., full-physics 6-DOF flight dynamics), the behavior must be reoptimized. To test this concept, we can change the rules of motion in a way that maintains low computational cost, using these different rules as a proxy for higher fidelity.

The original benchmark problem had grid-based rules of motion in which the glider could move only in the four compass directions at a speed of one unit of distance per one unit of time. In our simplest implementation of adaptive waypoints, we increase the number of degrees of freedom by allowing the gliders to move in any direction, but with a constraint on turning radius (or equivalently, on maximum turning angle per time step). As before, the glider speed is held constant at one unit per time step. Because the glider can now move in any direction, it can occupy any location within the simulation space. Its direction and coordinates are described by real numbers rather than by integers.

The “fidelity proxy” for this test is the maximum turning angle ( ). We performed a series of optimization runs with set to 30°. We then varied between 5° and 85° and recalculated the fitnesses of the trees. Three examples are plotted in Figure 9-4. Each of the three behaviors yields a fitness value near the theoretical maximum for = 30°, but shows strikingly different dependencies on . For example, the dependence of S01 is described by a narrow spike at 30°, suggesting that this behavior is not robust and is not a good candidate for incremental adaptation. The behavior exhibited by S03 falls off sharply for angles less than 30°, but remains high for larger angles, suggesting a range of adaptability. The most robust behavior is demonstrated by S02, which retains high fitness over a wide range of angles.

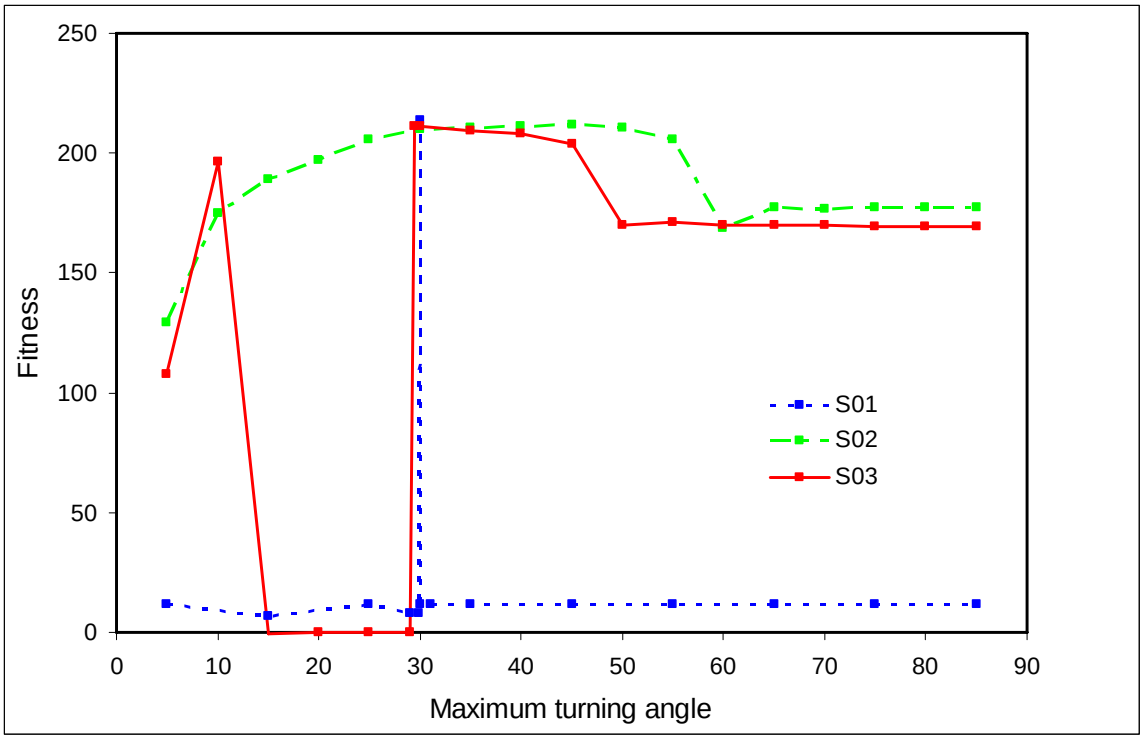


Figure 9-4. Fitness as a function of turning angle for three behaviors.

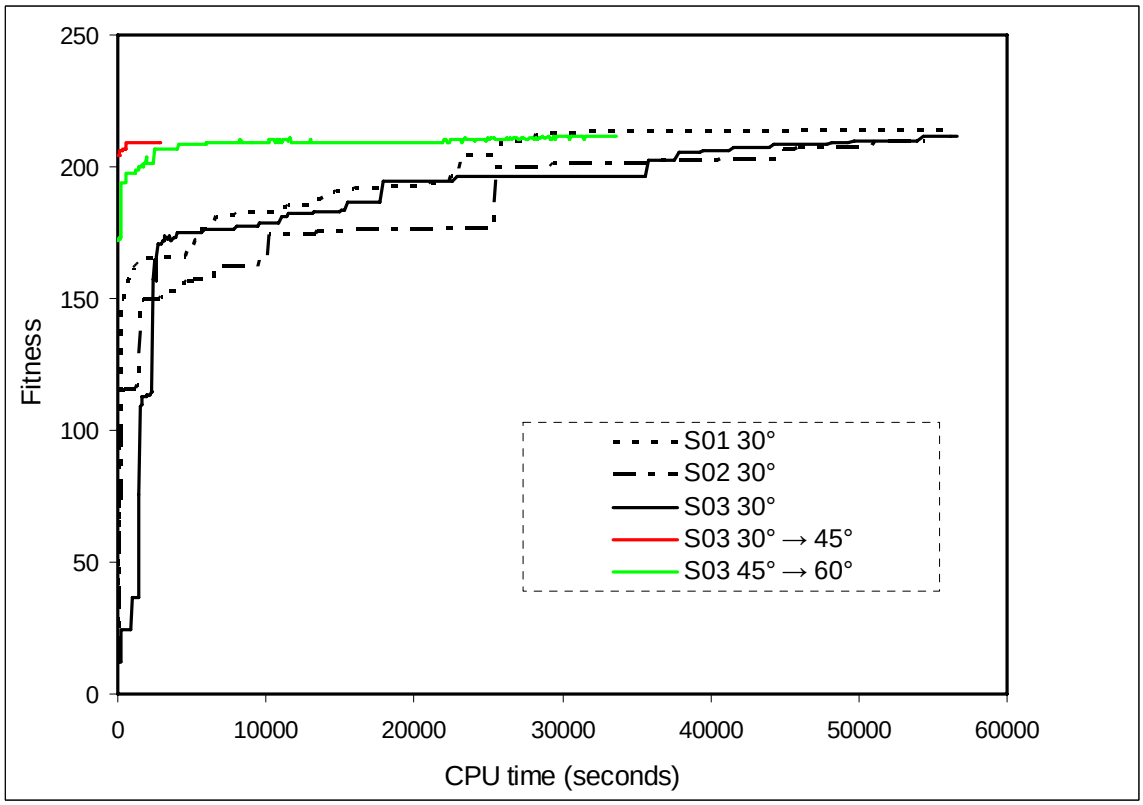


Figure 9-5. Fitness as a function of processor time for three behavior runs.

Figure 9-5 shows the fitness history of these three behaviors as they evolved on a single-processor desktop computer. These were run with small population sizes (n=40) so that the evolutionary process could be observed for the entire population during the simulation. The behaviors for this problem typically approached the theoretical maximum during an overnight run. To illustrate the effectiveness of incremental adaptation, we used the maximum turning angle as our "fidelity proxy," restarting the S03 evolution run with set to 45° (red curve, labeled SO3 30° → 45°). Within a few generations, the fitness had regained more than half the value it had lost compared to the original run. We again restarted the S03 evolution run with set to 60° (green curve, labeled SO3 45° → 60°). The behavior again readapted to the different rules of motion and rapidly regained the fitness it had lost. These and other results for similar runs support our assertion that incremental adaptation is one means by which scalability and wide applicability can be addressed when applying evolutionary methods.

10 Summary

We have introduced a hybrid AI approach that has many advantages for generating autonomous behaviors for mobile robots that operate in a complex, dynamic environment. This method is needed to generalize across different conditions because of the diversity of physical environments, vehicles, locomotion abilities, and interaction dynamics with other objects. We encapsulate units of behavior as building blocks that can either be evolved or human engineered and that can be combined in horizontal or vertical hierarchies.

We developed the specifications and implemented an object-oriented code called DGGP (Distributed Generic Genetic Programming) and integrated it with the Umbra modeling and simulation framework. This integrated software allows behaviors to be generated, evolved, and reused. It also means that the DGGP engine can take advantage of modules created by the Umbra community, leading to the potential for wider use and collaboration.

Our methodology can be applied to a wide range of applications far beyond goal-oriented navigation. We use a LOB (level-of-behavior) approach to achieve a balance between reactive and deliberative behaviors. We make use of incremental adaptation, staged optimization, and graduated embodiment.

Incremental adaptation is a method by which building blocks can be taken from one situation and reused in a slightly different one. By successive readaptation to a sequence of environments or situations, a behavior's use can be migrated and reoptimized for other purposes. Staged optimization is a means of breaking a complex task into smaller subtasks that can be independently optimized. This addresses the inherent nonscalability of evolutionary methods by optimizing components in succession, either independently or in coevolutionary sets. Combining these methods, for the special case of evolution to higher levels of fidelity and from the virtual to the real world, is our way of achieving graduated embodiment.

11 Future Work

The methods and tools developed for this project are directly applicable to other agent-based modeling needs at Sandia, and we are further enhancing the capabilities of the integrated DGGP-Umbra software. Our next goal is to integrate a recently developed genetic algorithm conditioner, GAPT, which uses a *parallel tempering* methodology (Slepoy and Thompson 2004) to improve the convergence times of genetic algorithms by maintaining diversity of the gene pool. We will investigate the use of GAPT to reduce the computational time of the DGGP-Umbra system on Sandia's large-scale parallel architectures. We have selected a site-security problem because of its importance to Sandia's security, the challenge of modeling human behavior in complex environments, and the combinatorial nature of the solution. Our plan is to develop a simulation capability to compute the fitness of potential solution genes that encode security strategies.

We are also in the process of developing an agent-based coupled climate model to assess climate-change effects on international stability (Boslough et al. 2004). Part of this project will be devoted to the development of an agent-based social model that can be used to study the sensitivity of social, political, and economic situations to climate change. The DGGP-Umbra framework will allow us to experiment with our agent models, for example, by evolving individual agent behaviors to optimize some emergent property of the collective.

There are several other projects in various stages of proposal and planning that can make use of these tools and methods. These include the creation of behaviors and strategies for nonplayer characters in multiplayer games for training exercises and intelligent after-action reviews, with direct application to military operations in urbanized terrain (MOUT). Behavior modeling and simulation software is also needed to address problems of interest to the intelligence community. Such models have been identified as necessary to make the connection between terrorist motivations and observable behavior, and to more accurately interpret such behaviors (e.g., recruitment, public imaging, and targeting strategies). Finally, we would like to apply these tools toward the development of trading agents and models for understanding and implementing information aggregation markets for evaluating hypotheses, with applications for the intelligence community.

References

Barnette, D. W., R. J. Pryor, and J. T. Feddema. 2000. *Development and Application of Genetic Algorithms for Sandia's RATLER Robotic Vehicles*. SAND2000-2846. Albuquerque, NM: Sandia National Laboratories.

Beard, R. W., and T. W. McLain. 2003. "Multiple UAV Cooperative Search under Collision Avoidance and Limited Range Communication Constraints." In *Proceedings of the 2003 IEEE Conference on Decision and Control*, 25–30.

Boslough, M. B. E. 2002. *Autonomous Dynamic Soaring Platform for Distributed Mobile Sensor Arrays*. SAND2002-1896. Albuquerque, NM: Sandia National Laboratories.

Boslough, M. B. E., et al. 2004. *Climate Change Effects on International Stability: A White Paper*. SAND2004-5973. Albuquerque, NM: Sandia National Laboratories.

Brooks, R. A. 1986. "A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation* RA-2(1): 14–23.

———. 1990. "Elephants Don't Play Chess." *Robotics and Autonomous Systems* 6, nos. 1–2 (June): 3–15.

———. 1991. "Intelligence Without Reason." In *Proceedings of the 1991 Int. Joint Conference on Artificial Intelligence*, 569–95.

Cliff, D. 1991. "Computational Neuroethology: A Provisional Manifesto." In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, edited by J. -A. Meyer and S. W. Wilson, 29–39. Cambridge, MA: MIT Press.

Dautenhahn, K., B. Ogden, and T. Quick. 2001. "From Embodied to Socially Embedded Agents – Implications for Interaction-Aware Robots." *Cognitive Systems Research* 3, no. 3:397–428 [Special issue on Situated and Embodied Cognition, guest editor: Tom Ziemke].

Duffy, B. R., and G. Joue. 2001. "Embodied Mobile Robots." In *Proceedings of the 1st International Conference on Autonomous Minirobots for Research and Edutainment – AMiRE2001*, Paderborn, Germany, October 22–25, 2001.

Filliat, D., and J. Meyer. 2003. "Map-Based Navigation in Mobile Robots. I. A Review of Localization Strategies." *Journal of Cognitive Systems Research*, in press.

Gat, E. 1997. "On Three-Layer Architectures." In *Artificial Intelligence and Mobile Robots*, edited by D. Kortenkamp, R. P. Bonasso, and R. Murphy. MIT/AAAI Press.

Holland, J. H. 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press.

Koza, J., F. Bennett III, D. Andre, and M. Keane. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving.* San Francisco, CA: Morgan Kaufmann.

Maes, P. 1993. "Behavior-Based Artificial Intelligence." In *From Animals to Animats: Proceedings of the Second International Conference on Simulation of Adaptive* Behavior, edited by J. -A. Meyer, H. Roitblat, and S. W. Wilson, 2–10. Cambridge, MA: MIT Press.

Meyer, J., and D. Filliat. 2003. "Map-Based Navigation in Mobile Robots. II. A Review of Map-Learning and Path-Planning Strategies." *Journal of Cognitive Systems Research*, in press.

Moravec, H. P. 1984. "Locomotion, Vision, and Intelligence." In *Robotics Research 1*, edited by M. Brady and R. Paul, 215–24. Cambridge, MA: MIT Press.

Peters, M., M. Boslough, and C. Jorgensen. 2005. *Distributed Generic Genetic Programming User's Guide*, in preparation. Albuquerque, NM: Sandia National Laboratories.

Pryor, R. J. 1998. *Developing Robotic Behavior Using a Genetic Programming Model*. SAND98-0074. Albuquerque, NM: Sandia National Laboratories.

Pryor, R. J., and D. Barton. 2002. *Developing Maneuvering Behaviors for a Glider UAV Using a Genetic Programming Model*. SAND2002-3147. Albuquerque, NM: Sandia National Laboratories.

Sharkey, N., and T. Ziemke. 2000. "Life, Mind and Robots: The Ins and Outs of Embodied Cognition." In *Hybrid Neural Systems*, edited by S. Wermter and R. Sun, 313–32. Heidelberg: Springer Verlag.

Sims, K. 1994. "Evolving 3D Morphology and Behavior by Competition." *Artificial Life* 1, no. 4 (Summer): 353–72.

Slepoy, A., and A. P. Thompson. 2004. "Illustration of Dramatic Convergence Acceleration of Genetic Algorithm Using Parallel Tempering," unpublished report. Albuquerque, NM: Sandia National Laboratories.

Weiss, G., ed. 1999. *Multiagent Systems.* Cambridge, MA: MIT Press.

Wilson, S. W. 1985. "Knowledge Growth in an Artificial Animal. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, 16–23. Hillsdale, NJ: Lawrence Erlbaum Associates.

Distribution

1	MS 0123	H. R. Westrich	01011
1	MS 0196	J. A. Sprigg	09216
1	MS 0196	G. A. Backus	09216
1	MS 0316	S. S. Dosanjh	09233
1	MS 0318	G. S. Davidson	09200
5	MS 0318	J. E. Nelson	09216
20	MS 0318	S. G. Wagner	09209
20	MS 0318	M. B. E. Boslough	09216
10	MS 0318	M. D. Peters	09216
1	MS 0318	C. R. Jorgensen	09216
1	MS 0321	W. J. Camp	09200
1	MS 0321	R. W. Leland	09220
1	MS 0735	T. S. Lowry	06115
1	MS 0776	J. S. Stein	06852
1	MS 0826	R. K. Reinert	10254
1	MS 0839	N. K. Hayden	05925
1	MS 1002	P. D. Heermann	15230
1	MS 1003	S. C. Roehrig	15200
1	MS 1004	A. R. Pierson	15231
1	MS 1004	F. J. Oppel, III	15231
1	MS 1109	R. J. Pryor	09216
1	MS 1110	D. E. Womble	09210
1	MS 1110	A. Slepoy	09235
1	MS 1138	M. A. Ehlen	06221
1	MS 1188	J. S. Wagner	15241
1	MS 1188	E. M. Raybourn	15241
1	MS 1377	M. J. McDonald	06956
1	MS 9018	Central Technical Files	08945-1
2	MS 0899	Technical Library	09616
1	MS 0612	Review & Approval Desk for DOE/OSTI	09612